E. Allen Emerson
Kedar S. Namjoshi (Eds.)

LNCS 3855

Verification, Model Checking, and Abstract Interpretation

7th International Conference, VMCAI 2006
Charleston, SC, USA, January 2006
Proceedings

Lecture Notes in Computer Science 3855

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
New York University, NY, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

E. Allen Emerson Kedar S. Namjoshi (Eds.)

Verification, Model Checking, and Abstract Interpretation

7th International Conference, VMCAI 2006
Charleston, SC, USA, January 8-10, 2006
Proceedings

Springer

Volume Editors

E. Allen Emerson
University of Texas at Austin, Department of Computer Science
Austin, TX 78712, USA
E-mail: emerson@cs.utexas.edu

Kedar S. Namjoshi
Bell Labs, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974, USA
E-mail: kedar@research.bell-labs.com

Library of Congress Control Number: 2005937944

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 3-540-31139-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-31139-3 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11609773 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers accepted for presentation at the 7th International Conference on Verification, Model Checking, and Abstract Interpretation, held January 8-10, 2006, at Charleston, South Carolina, USA.

VMCAI provides a forum for researchers from the communities of verification, model checking, and abstract interpretation, facilitating interaction, cross-fertilization, and advancement of hybrid methods.

The program was selected from 58 submitted papers. In addition, the program included invited talks by Edmund M. Clarke (Carnegie Mellon University), James R. Larus (Microsoft Research), and Greg Morrisett (Harvard University), and invited tutorials by Nicolas Halbwachs (VERIMAG) and David Schmidt (Kansas State University).

VMCAI was sponsored by the University of Texas at Austin, with additional support from Microsoft Research and NEC Research Labs. We are grateful for the support. We would like to thank the Program Committee and the reviewers for their hard work and dedication in putting together this program. We especially wish to thank Jacob Abraham, director of the Computer Engineering Research Center at the University of Texas at Austin and Debi Prather for their invaluable support and assistance, and Rich Gerber for his help with the START conference management system.

January 2006

E. Allen Emerson
Kedar S. Namjoshi

Organization

Program Committee

Alex Aiken	Stanford University, USA
Thomas Ball	Microsoft Research, USA
Hana Chockler	IBM Research, Israel
Patrick Cousot	École Normale Supérieure, France
E. Allen Emerson (Co-chair)	University of Texas at Austin, USA
Javier Esparza	University of Stuttgart, Germany
Roberto Giacobazzi	University of Verona, Italy
Patrice Godefroid	Bell Laboratories, USA
Warren Hunt	University of Texas at Austin, USA
Neil Jones	DIKU, University of Copenhagen, Denmark
Tiziana Margaria	University of Göttingen, Germany
Markus Müller-Olm	University of Münster, Germany
Kedar S. Namjoshi (Co-chair)	Bell Laboratories, USA
George Necula	University of California at Berkeley, USA
Jens Palsberg	UCLA, USA
Andreas Podelski	Max-Planck-Institut für Informatik, Germany
Thomas W. Reps	University of Wisconsin, USA
A. Prasad Sistla	University of Illinois at Chicago, USA
Colin Stirling	University of Edinburgh, UK
Scott D. Stoller	SUNY at Stony Brook, USA
Lenore Zuck	University of Illinois at Chicago, USA

Steering Committee

Agostino Cortesi	University of Venice, Italy
Patrick Cousot	École Normale Supérieure, France
E. Allen Emerson	University of Texas at Austin, USA
Giorgio Levi	University of Pisa, Italy
Andreas Podelski	Max-Planck-Institut für Informatik, Germany
Thomas W. Reps	University of Wisconsin, USA
David Schmidt	Kansas State University, USA
Lenore Zuck	University of Illinois at Chicago, USA

Sponsoring Institutions

The University of Texas at Austin
Microsoft Research
NEC Research Labs

Referees

Parosh Abdulla
Gilad Arnold
Gadiel Auerbach
James Avery
Bernard Boigelot
Ahmed Bouajjani
Glenn Bruns
Scott Cotton
Dennis Dams
Jared Davis
John Erickson
Xiang Fu
Maria del Mar Gallardo
Peter Habermehl
Brian Hackett
Tom Henzinger
Bertrand Jeannet
Robert Krug
Viktor Kuncak
Orna Kupferman
Alexander Malkis
Damien Masse
Isabella Mastroeni
Alessio Merlo
David Monniaux
Anders Møller
Serita Neleson
Ziv Nevo
Tobias Nipkow
Avigail Orni
Julien d'Orso
German Puebla
Mila Dalla Preda
David Rager
C.R. Ramakrishnan
Francesco Ranzato
Sandip Ray
Erik Reeber
Xavier Rival
Oliver Rüthing
Giovanni Scardoni
Stefan Schwoon
Roberto Segala
Zhong Shao
Jakob Grue Simonsen
Viorica Sofronie-Stokkermans
Fausto Spoto
Bernhard Steffen
Sol Swords
Wenkai Tan
Francesco Tapparo
Tachio Terauchi
Helmut Veith
Mahesh Viswanathan
Hubert Wagner
Thomas Wies
Daniel Wilkerson
Ping Yang
Pei Ye
Karen Yorav
Greta Yorsh
Damiano Zanardini
Qiang Zhang
Min Zhou

Table of Contents

Closure Operators for ROBDDs
Peter Schachte, Harald Søndergaard 1

A CLP Method for Compositional and Intermittent Predicate Abstraction
Joxan Jaffar, Andrew E. Santosa, Răzvan Voicu 17

Combining Shape Analyses by Intersecting Abstractions
Gilad Arnold, Roman Manevich, Mooly Sagiv, Ran Shaham 33

A Complete Abstract Interpretation Framework for Coverability Properties of WSTS
Pierre Ganty, Jean-François Raskin, Laurent Van Begin 49

Complexity Results on Branching-Time Pushdown Model Checking
Laura Bozzelli 65

A Compositional Logic for Control Flow
Gang Tan, Andrew W. Appel 80

Detecting Non-cyclicity by Abstract Compilation into Boolean Functions
Stefano Rossignoli, Fausto Spoto 95

Efficient Strongly Relational Polyhedral Analysis
Sriram Sankaranarayanan, Michael A. Colón, Henny B. Sipma, Zohar Manna 111

Environment Abstraction for Parameterized Verification
Edmund Clarke, Muralidhar Talupur, Helmut Veith 126

Error Control for Probabilistic Model Checking
Håkan L.S. Younes 142

Field Constraint Analysis
Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, Martin Rinard 157

A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety
Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula 174

Improved Algorithm Complexities for Linear Temporal Logic Model Checking of Pushdown Systems
Katia Hristova, Yanhong A. Liu 190

A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs
Jesse Bingham, Zvonimir Rakamarić 207

Monitoring Off-the-Shelf Components
A. Prasad Sistla, Min Zhou, Lenore D. Zuck 222

Parallel External Directed Model Checking with Linear I/O
Shahid Jabbar, Stefan Edelkamp 237

Piecewise FIFO Channels Are Analyzable
Naghmeh Ghafari, Richard Trefler 252

Ranking Abstraction of Recursive Programs
Ittai Balaban, Ariel Cohen, Amir Pnueli 267

Relative Safety
Joxan Jaffar, Andrew E. Santosa, Răzvan Voicu 282

Resource Usage Analysis for the π-Calculus
Naoki Kobayashi, Kohei Suenaga, Lucian Wischik 298

Semantic Hierarchy Refactoring by Abstract Interpretation
Francesco Logozzo, Agostino Cortesi 313

Strong Preservation of Temporal Fixpoint-Based Operators by Abstract Interpretation
Francesco Ranzato, Francesco Tapparo 332

Symbolic Methods to Enhance the Precision of Numerical Abstract Domains
Antoine Miné 348

Synthesis of Reactive(1) Designs
Nir Piterman, Amir Pnueli, Yaniv Sa'ar 364

Systematic Construction of Abstractions for Model-Checking
Arie Gurfinkel, Ou Wei, Marsha Chechik 381

Totally Clairvoyant Scheduling with Relative Timing Constraints
K. Subramani 398

Verification of Well-Formed Communicating Recursive State Machines
Laura Bozzelli, Salvatore La Torre, Adriano Peron 412

What's Decidable About Arrays?
Aaron R. Bradley, Zohar Manna, Henny B. Sipma 427

Author Index 443

Closure Operators for ROBDDs

Peter Schachte* and Harald Søndergaard

Department of Computer Science and Software Engineering,
The University of Melbourne, Vic. 3010, Australia

Abstract. Program analysis commonly makes use of Boolean functions to express information about run-time states. Many important classes of Boolean functions used this way, such as the monotone functions and the Boolean Horn functions, have simple semantic characterisations. They also have well-known syntactic characterisations in terms of Boolean formulae, say, in conjunctive normal form. Here we are concerned with characterisations using binary decision diagrams. Over the last decade, ROBDDs have become popular as representations of Boolean functions, mainly for their algorithmic properties. Assuming ROBDDs as representation, we address the following problems: Given a function ψ and a class of functions Δ, how to find the strongest $\varphi \in \Delta$ entailed by ψ (when such a φ is known to exist)? How to find the weakest $\varphi \in \Delta$ that entails ψ? How to determine that a function ψ belongs to a class Δ? Answers are important, not only for several program analyses, but for other areas of computer science, where Boolean approximation is used. We give, for many commonly used classes Δ of Boolean functions, algorithms to approximate functions represented as ROBDDs, in the sense described above. The algorithms implement upper closure operators, familiar from abstract interpretation. They immediately lead to algorithms for deciding class membership.

1 Introduction

Propositional logic is of fundamental importance to computer science. While its primary use has been within switching theory, there are many other uses, for example in verification, machine learning, cryptography and program analysis. In complexity theory, Boolean satisfiability has played a seminal role and provided deep and valuable results.

Our own interest in Boolean functions stems from work in program analysis. In this area, as in many other practical applications of propositional logic, we are not so much interested in solving Boolean equations, as in using Boolean functions to capture properties and relations of interest. In the process of analysing programs, we build and transform representations of Boolean functions, in order to provide detailed information about runtime states.

In this paper we consider various instances of the following problem. Given a Boolean function φ and a class of Boolean functions Δ, how can one decide

* Peter Schachte's work has been supported in part by NICTA Victoria Laboratories.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 1–16, 2006.
© Springer-Verlag Berlin Heidelberg 2006

(efficiently) whether φ belongs to Δ? How does one find the strongest statement in Δ which is entailed by φ (assuming this is well defined)? Answers of course depend on how Boolean functions are represented.

ROBDDs [4] provide a graphical representation of Boolean functions, based on repeated Boolean development, that is, the principle that in any Boolean algebra, $\varphi = (\overline{x} \wedge \varphi_x^0) \vee (x \wedge \varphi_x^1)$ where φ_x^u denotes φ with x fixed to the truth value u. In this paper, ROBDDs are used to represent Boolean functions.

The classes of Boolean functions studied here are classes that have simple syntactic and semantic characterisations. They are are of importance in many areas of computer science, including program analysis. They include the Horn fragment, monotone and antitone Boolean functions, sub-classes of bijunctive functions, and others.

There are many examples of the use of approximation in program analysis. Trivial cases are where a program analysis tool uses intermediate results that are of a finer granularity that what gets reported to a user or used by an optimising compiler. Consider for example classical two-valued strictness analysis [18]. The strictness result for the functional program

```
g(x,y,z) = if even(x) then y/2 else 3*z + 1
```

is calculated to be $x \wedge (y \vee z)$. This contains a disjunctive component indicating that `g` needs at least one of its last two arguments, in addition to the first. This disjunctive information is not useful for a compiler seeking to replace call-by-name by call-by-value—instead of $x \wedge (y \vee z)$ a compiler needs the weaker statement x. (Once we have the definitions of Section 3 we can say that what is needed is the strongest **V** consequence of the calculated result.) That the more fine-grained $x \wedge (y \vee z)$ is useful as an intermediate result, however, becomes clear when we consider the function

```
f(u,v) = g(u,v,v)
```

whose strictness result is $u \wedge (v \vee v)$, that is, $u \wedge v$. Without the disjunctive component in `g`'s result, the result for `f` would be unnecessarily weak.

Less trivial cases are when approximation is needed in intermediate results, to guarantee correctness of the analysis. Genaim and King's suspension analysis for logic programs with dynamic scheduling [9] is an example. The analysis, essentially a greatest-fixed-point computation, produces for each predicate p a Boolean formula φ_i expressing groundness conditions under which the atomic formulae in the body of p may be scheduled so as to obtain suspension-free evaluation. In each iteration, the re-calculation of the formulae φ includes a crucial step where φ is replaced by its weakest monotone implicant. Similarly, set-sharing analysis as presented by Codish *et al.* [5] relies on an operation that replaces a positive formula by its strongest definite consequence.

The contribution of the present paper is two-fold. First, we view a range of important classes of Boolean functions from a new angle. Studying these classes of Boolean functions through the prism of Boolean development (provided by ROBDDs) yields deeper insight both into ROBDDs and the classes themselves.

The first three sections of this paper should therefore be of value to anybody with an interest in the theory of Boolean functions. Second, we give algorithms for ROBDD approximation. The algorithms are novel, pleasingly simple, and follow a common pattern. This more practical contribution may be of interest to anybody who works with ROBDDs, but in particular to those who employ some kind of approximation of Boolean functions, as it happens for example in areas of program analysis, cryptography, machine learning and property testing.

The reader is assumed to be familiar with propositional logic and ROBDDs. Section 2 recapitulates ROBDDs and some standard operations, albeit mainly to establish our notation. In Section 3 we define several classes of Boolean functions and establish, for each, relevant properties possessed by their members. Section 4 presents new algorithms for approximating Boolean functions represented as ROBDDs. The aim is to give each method a simple presentation that shows its essence and facilitates a proof of correctness. Section 5 discusses related work and Section 6 concludes.

2 ROBDDs

We briefly recall the essentials of ROBDDs [4]. Let the set $\mathcal{V}$ of propositional variables be equipped with a total ordering $\prec$. *Binary decision diagrams* (*BDDs*) are defined inductively as follows:

- 0 is a BDD.
- 1 is a BDD.
- If $x \in \mathcal{V}$ and R_1 and R_2 are BDDs then $\mathsf{ite}(x, R_1, R_2)$ is a BDD.

Let $R = \mathsf{ite}(x, R_1, R_2)$. We say a BDD R' *appears in* R iff $R' = R$ or R' appears in R_1 or R_2. We define $\mathsf{vars}(R) = \{v \mid \mathsf{ite}(v, _, _) \text{ appears in } R\}$. The meaning of a BDD is given as follows.

$$\begin{aligned} [\![0]\!] &= \mathit{0} \\ [\![1]\!] &= \mathit{1} \\ [\![\mathsf{ite}(x, R_1, R_2)]\!] &= (x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]) \end{aligned}$$

A BDD is an *OBDD* iff it is 0 or 1 or if it is $\mathsf{ite}(x, R_1, R_2)$, R_1 and R_2 are OBDDs, and $\forall x' \in \mathsf{vars}(R_1) \cup \mathsf{vars}(R_2) : x \prec x'$.

An OBDD R is an *ROBDD* (Reduced Ordered Binary Decision Diagram, [3, 4]) iff for all BDDs R_1 and R_2 appearing in R, $R_1 = R_2$ when $[\![R_1]\!] = [\![R_2]\!]$. Practical implementations [2] use a function $\mathsf{mknd}(x, R_1, R_2)$ to create all ROBDD nodes as follows:

1. If $R_1 = R_2$, return R_1 instead of a new node, as $[\![\mathsf{ite}(x, R_1, R_2)]\!] = [\![R_1]\!]$.
2. If an identical ROBDD was previously built, return that one instead of a new one; this is accomplished by keeping a hash table, called the *unique table*, of all previously created nodes.
3. Otherwise, return $\mathsf{ite}(x, R_1, R_2)$.

This ensures that ROBDDs are strongly canonical: a shallow equality test is sufficient to determine whether two ROBDDs represent the same Boolean function.

Figure 1 shows an example ROBDD in diagrammatic form. This ROBDD denotes the function $(x \leftarrow y) \rightarrow z$. An ROBDD $\mathsf{ite}(x, R_1, R_2)$ is depicted as a directed acyclic graph rooted in x, with a solid arc from x to the dag for R_1 and a dashed line from x to the dag for R_2.

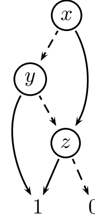


Fig. 1. $(x \leftarrow y) \rightarrow z$

It is important to take advantage of *fan-in* to create efficient ROBDD algorithms. Often some ROBDD nodes will appear multiple times in a given ROBDD, and algorithms that traverse that ROBDD will meet these nodes multiple times. Many algorithms can avoid repeated work by keeping a cache of previously seen inputs and their corresponding outputs, called a *computed table*. See Brace *et al.* [2] for details. We assume a computed table is used for all recursive ROBDD algorithms presented here.

3 Boolean Function Classes as Closure Operators

Let $\mathcal{B} = \{\mathit{0}, \mathit{1}\}$. A Boolean function over variable set $\mathcal{V} = \{x_1, \ldots, x_n\}$ is a function $\varphi : \mathcal{B}^n \rightarrow \mathcal{B}$. We assume a fixed, finite set $\mathcal{V}$ of variables, and use $\mathbf{B}$ to denote the set of all Boolean functions over $\mathcal{V}$. The ordering on $\mathcal{B}$ is the usual: $x \leq y$ iff $x = \mathit{0} \lor y = \mathit{1}$. $\mathbf{B}$ is ordered pointwise, that is, the ordering relation is entailment, $\models$.

The class $\mathbf{C} \subset \mathbf{B}$ contains just the two constant functions. As is common, we use '$\mathit{0}$' and '$\mathit{1}$' not only to denote the elements of $\mathcal{B}$, but also for the elements of $\mathbf{C}$. The class $\mathbf{1} \subset \mathbf{C}$, contains only the element $\mathit{1}$.

A *valuation* $\mu : \mathcal{V} \rightarrow \mathcal{B}$ is an assignment of truth values to the variables in $\mathcal{V}$. Valuations are ordered pointwise. We will sometimes write a valuation as the set of variables which are assigned the value $\mathit{1}$. In this view, the *meet* operation on valuations is set intersection, and the *join* is set union.

A valuation μ is a *model* for φ, denoted $\mu \models \varphi$, if $\varphi(\mu(x_1), \ldots, \mu(x_n)) = \mathit{1}$. In the "set" view, the set of models of φ is a set of sets of variables, namely

$$[\![\varphi]\!]_{\mathcal{V}} = \left\{ \{x \in \mathcal{V} \mid \mu(x) = \mathit{1}\} \,\middle|\, \mu \models \varphi \right\}.$$

Again, we will often omit the subscript $\mathcal{V}$ as it will be clear from the context. We will also switch freely amongst the views of φ as a function, a formula, and as its set of models, relying on the reader to disambiguate from context.

We say that a Boolean function φ is *model-meet closed* if, whenever $\mu \models \varphi$ and $\mu' \models \varphi$, we also have $\mu \cap \mu' \models \varphi$. In other words, φ is model-meet closed if $[\![\varphi]\!]$ is closed under intersection. Similarly, φ is *model-join closed* if, whenever $\mu \models \varphi$ and $\mu' \models \varphi$, also $\mu \cup \mu' \models \varphi$. We likewise say that a Boolean function φ is *downward closed* if, whenever $\mu \models \varphi$, we also have $\mu \cap \mu' \models \varphi$ for all valuations μ', and similarly φ is *upward closed* if, whenever $\mu \models \varphi$, we also have $\mu \cup \mu' \models \varphi$ for all valuations μ'. Note that a downward closed function is necessarily model-meet closed, and an upward closed function is model-join closed.

Let $\overline{\mathcal{V}} = \{\overline{x} \mid x \in \mathcal{V}\}$ be the set of negated variables. A *literal* is a member of the set $\mathcal{L} = \mathcal{V} \cup \overline{\mathcal{V}}$, that is, a variable or a negated variable. We say that φ is *independent* of literal x (and also of literal $\overline{x}$) when for all models μ of φ, $\mu \setminus \{x\} \models \varphi$ iff $\mu \cup \{x\} \models \varphi$.

The *dual* of a Boolean function φ is the function that is obtained by interchanging the roles of *1* and *0*. A simple way of turning a formula for φ into a formula for φ's dual is to change the sign of every literal in φ and negate the whole resulting formula. For example, the dual of $x \wedge (\overline{y} \vee z)$ is $x \vee (\overline{y} \wedge z)$ — De Morgan's laws can be regarded as duality laws.

Define φ° as the dual of $\overline{\varphi}$. Following Halmos [13], we call φ° the *contra-dual* of φ. Clearly, given a formula for φ, a formula for φ° is obtained by changing the sign of each literal in φ. As an example, $((x \leftrightarrow y) \rightarrow z)^\circ = (x \leftrightarrow y) \rightarrow \overline{z}$. Alternatively, given a truth table for a Boolean function, the truth table for its contra-dual is obtained by turning the result column upside down. Given an ROBDD R for φ, we can also talk about R's contra-dual, R°, which represents φ°. An ROBDD's contra-dual is obtained by simultaneously making all solid arcs dashed, and all dashed arcs solid.

Clearly the mapping $\varphi \mapsto \varphi^\circ$ is an involution, and monotone: $\psi \models \varphi$ iff $\psi^\circ \models \varphi^\circ$. Note that φ° is model-join closed iff φ is model-meet closed. For any class Δ of Boolean functions, we let Δ° denote the class $\{\varphi^\circ \mid \varphi \in \Delta\}$.

The classes of Boolean functions considered in this paper can all be seen as upper closures of $\mathbf{B}$. Recall that an *upper closure operator* (or just *uco*) $\rho : L \rightarrow L$ on a complete lattice L satisfies the following constraints:

- It is monotone: $x \sqsubseteq y$ implies $\rho(x) \sqsubseteq \rho(y)$ for all $x, y \in L$.
- It is extensive: $x \sqsubseteq \rho(x)$ for all $x \in L$.
- It is idempotent: $\rho(x) = \rho(\rho(x))$ for all $x \in L$.

As each class Δ under study contains *1* and is closed under conjunction (this will be obvious from the syntactic characterisations given below), Δ is a lattice. Moreover, the mapping $\rho_\Delta : \mathbf{B} \rightarrow \mathbf{B}$, defined by

$$\rho_\Delta(\psi) = \bigwedge\{\varphi \in \Delta \mid \psi \models \varphi\}$$

is a uco on $\mathbf{B}$. Since it is completely determined by Δ, and vice versa, we will usually denote ρ_Δ simply as Δ. The view of such classes (*abstract domains*) Δ as upper closure operators has been popular ever since the seminal papers on abstract interpretation [6, 7].

We list some well-known properties of closure operators [23, 6]. Let L be a complete lattice $(L, \bot, \top, \bigsqcap, \bigsqcup)$ and let $\rho : L \rightarrow L$ be a uco. Then $\rho(L)$ is a complete lattice $(\rho(L), \rho(\bot), \top, \bigsqcap, \lambda S.\rho(\bigsqcup S))$. It is a complete sublattice of L if and only if ρ is additive, that is, $\rho(\bigsqcup S) = \bigsqcup \rho(S)$ for all $S \subseteq L$. In any case,

$$\rho(\bigsqcap S) \sqsubseteq \bigsqcap \rho(S) = \rho(\bigsqcap \rho(S)) \tag{1}$$

$$\bigsqcup \rho(S) \sqsubseteq \rho(\bigsqcup S) = \rho(\bigsqcup \rho(S)) \tag{2}$$

Given two upper closure operators ρ and ρ' on the same lattice L, $\rho \circ \rho'$ need not be an upper closure operator. However, if $\rho \circ \rho' = \rho' \circ \rho$ then the composition is also an upper closure operator, and $\rho(\rho'(L)) = \rho'(\rho(L)) = \rho(L) \cap \rho'(L)$ [10, 19].

The Boolean function classes we focus on here are those characterised by combinations of certain interesting semantic properties: model-meet closure, model-join closure, downward closure, and upward closure. Nine classes are spanned, as shown in the Hasse diagram of Figure 2. These classes are chosen for their importance in program analysis, although our method applies to many other natural classes, as we argue in Section 5.

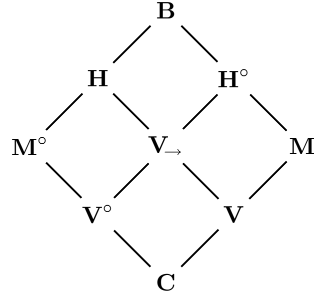


Fig. 2. Boolean function classes

We define $\mathbf{H}$ to be the set of model-meet closed Boolean functions, so $\mathbf{H}^\circ$ is the set of model-join closed functions. Similarly, we define $\mathbf{M}$ to be the upward-closed functions, with $\mathbf{M}^\circ$ the downward-closed functions. We define $\mathbf{V}_\rightarrow$ to be the set of Boolean functions that are both model-meet and model-join closed; *i.e.*, $\mathbf{H} \cap \mathbf{H}^\circ$. In what follows we utilise that $\mathbf{V}_\rightarrow$ is a uco, and therefore $\mathbf{H} \circ \mathbf{H}^\circ = \mathbf{H}^\circ \circ \mathbf{H} = \mathbf{V}_\rightarrow$. We also define $\mathbf{V} = \mathbf{H} \cap \mathbf{M} = \mathbf{V}_\rightarrow \cap \mathbf{M}$ and $\mathbf{V}^\circ = \mathbf{H}^\circ \cap \mathbf{M}^\circ = \mathbf{V}_\rightarrow \cap \mathbf{M}^\circ$, both of which are ucos, as well. Finally, we observe that $\mathbf{C} = \mathbf{M} \cap \mathbf{M}^\circ = \mathbf{V} \cap \mathbf{V}^\circ$ is also a uco. One can think of these elements as classes of functions, ordered by subset ordering, or alternatively, as upper closure operators, ordered pointwise (in particular, $\mathbf{B}$ is the identity function, providing loss-less approximation).

These classes (ucos) have a number of properties in common. All are closed under conjunction and existential quantification. None are closed under negation, and hence none are closed under universal quantification. All are closed under the operation of fixing a variable to a truth value. Namely, we can express instantiation using only conjunction and existential quantification. We write the fixing of x in φ to *0* as $\varphi_x^{\mathit{0}} \equiv \exists x : \varphi \wedge \overline{x}$ and the fixing of x in φ to *1* as $\varphi_x^{\mathit{1}} \equiv \exists x : \varphi \wedge x$. Finally, all of these classes enjoy a property that is essential to our algorithms: they do not introduce variables. For each uco Δ considered and each $x \in \mathcal{V}$ and $\varphi \in \mathbf{B}$, if φ is independent of x, then so is $\Delta(\varphi)$. In Section 5 we discuss a case where this property fails to hold.

We now define the various classes formally and establish some results that are essential in establishing the correctness of the algorithms given in Section 4.

3.1 The Classes M and M°

The class $\mathbf{M}$ of *monotone* functions consists of the functions φ satisfying the following requirement: for all valuations μ and μ', $\mu \cup \mu' \models \varphi$ when $\mu \models \varphi$. (These functions are also referred to as *isotone*.) Syntactically the class is most conveniently described as the the class of functions generated by $\{\wedge, \vee, \mathit{0}, \mathit{1}\}$, see for example Rudeanu's [21] Theorem 11.3. It follows that the uco $\mathbf{M}$ is additive.

The class $\mathbf{M}^\circ = \{\varphi^\circ \mid \varphi \in \mathbf{M}\}$ consists of the *antitone* Boolean functions. Functions φ in this class have the property that, for all valuations μ and μ', if $\mu \models \varphi$, then $\mu \cap \mu' \models \varphi$. $\mathbf{M}^\circ$, too, is additive.

As ROBDDs are based on the idea of repeated Boolean development, we are particularly interested in characterising class membership for formulae of the forms $x \vee \varphi$ and $\overline{x} \vee \varphi$ (with φ independent of x).

Lemma 1. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$. Then

(a) $x \vee \varphi \in \mathbf{M}$ iff $\varphi \in \mathbf{M}$ (c) $x \vee \varphi \in \mathbf{M}^\circ$ iff $\varphi \in \mathbf{1}$
(b) $\overline{x} \vee \varphi \in \mathbf{M}$ iff $\varphi \in \mathbf{1}$ (d) $\overline{x} \vee \varphi \in \mathbf{M}^\circ$ iff $\varphi \in \mathbf{M}^\circ$

Proof: In all cases, the 'if' direction is obvious from the well-known syntactic characterisations of $\mathbf{M}$ and $\mathbf{M}^\circ$. We show the 'only if' direction for cases (a) and (b); the proofs for (c) and (d) are similar.

(a) Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$, such that $x \vee \varphi \in \mathbf{M}$. Consider a model μ of φ. Since φ is independent of x, we have that $\mu \setminus \{x\} \models x \vee \varphi$. Let μ' be an arbitrary valuation. Then $(\mu \setminus \{x\}) \cup (\mu' \setminus \{x\}) \models x \vee \varphi$, so $(\mu \cup \mu') \setminus \{x\} \models \varphi$. Thus $\mu \cup \mu' \models \varphi$, and since μ' was arbitrary, $\varphi \in \mathbf{M}$.

(b) Suppose $\overline{x} \vee \varphi \in \mathbf{M}$. We show that every valuation is a model of φ. For any valuation μ, $\mu \setminus \{x\} \models \overline{x} \vee \varphi$. But then, $\mu \cup \{x\} \models \overline{x} \vee \varphi$, as $\overline{x} \vee \varphi \in \mathbf{M}$. As φ is independent of x, $\mu \models \varphi$. But μ was arbitrary, so φ must be *1*. ∎

3.2 The Classes H and H°

The class $\mathbf{H}$ of propositional Horn functions is exactly the set of model-meet closed Boolean functions. That is, every $\mathbf{H}$ function φ satisfies the following requirement: for all valuations μ and μ', if $\mu \models \varphi$ and $\mu' \models \varphi$, then $\mu \cap \mu' \models \varphi$. Similarly, $\mathbf{H}^\circ$ is the set of model-join closed Boolean functions, satisfying the requirement that for all valuations μ and μ', if $\mu \models \varphi$ and $\mu' \models \varphi$, then $\mu \cup \mu' \models \varphi$.

There are well-known syntactic characterisations of these classes. $\mathbf{H}$ is the set of functions that can be written in conjunctive normal form $\bigwedge(\ell_1 \vee \cdots \vee \ell_n)$ with at most one positive literal ℓ per clause, while $\mathbf{H}^\circ$ functions can be written in conjunctive normal form with each clause containing at most one *negative* literal.[1] It is immediate that $\mathbf{M} \subseteq \mathbf{H}^\circ$ and $\mathbf{M}^\circ \subseteq \mathbf{H}$.

The next lemma characterises membership of $\mathbf{H}$ and $\mathbf{H}^\circ$, for the case $\ell \vee \varphi$.

Lemma 2. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$. Then

(a) $x \vee \varphi \in \mathbf{H}$ iff $\varphi \in \mathbf{M}^\circ$ (c) $x \vee \varphi \in \mathbf{H}^\circ$ iff $\varphi \in \mathbf{H}^\circ$
(b) $\overline{x} \vee \varphi \in \mathbf{H}$ iff $\varphi \in \mathbf{H}$ (d) $\overline{x} \vee \varphi \in \mathbf{H}^\circ$ iff $\varphi \in \mathbf{M}$

Proof: In all cases, the 'if' direction follows easily from the syntactic characterisations of the classes. We prove the 'only if' directions for (a) and (b), as (c) and (d) are similar.

[1] An unfortunate variety of nomenclatures is used in Boolean taxonomy. For example, Schaefer [22] uses "weakly negative" for $\mathbf{H}$ and "weakly positive" for $\mathbf{H}^\circ$. Ekin *et al.* [8] use the term "Horn" to refer to $\{\overline{\varphi} \mid \varphi \in \mathbf{H}\}$ and "positive" for $\mathbf{M}$, while we use the word "positive" to refer to another class entirely (see Section 5).

(a) Assume $x \vee \varphi \in \mathbf{H}$ and x is independent of φ. Let μ be a model for φ and let μ' be an arbitrary valuation. Both $\mu \setminus \{x\}$ and $\mu' \cup \{x\}$ are models for $x \vee \varphi$. As $x \vee \varphi \in \mathbf{H}$, their intersection is a model as well, that is, $(\mu \cap \mu') \setminus \{x\} \models x \vee \varphi$. But then $(\mu \cap \mu') \models x \vee \varphi$, and as μ' was arbitrary, it follows that $\varphi \in \mathbf{M}^{\circ}$.

(b) Assume $\overline{x} \vee \varphi \in \mathbf{H}$ and x is independent of φ. Consider models μ and μ' for φ. As φ is independent of x, $\mu \setminus \{x\}$ and $\mu' \setminus \{x\}$ are models for $\overline{x} \vee \varphi$, and so $(\mu \setminus \{x\}) \cap (\mu' \setminus \{x\}) \models \overline{x} \vee \varphi$. But then $(\mu \setminus \{x\}) \cap (\mu' \setminus \{x\}) \models \varphi$, hence $\mu \cap \mu' \models \varphi$, so $\varphi \in \mathbf{H}$. ∎

3.3 The Class $\mathbf{V}_{\rightarrow}$

We define $\mathbf{V}_{\rightarrow} = \mathbf{H} \cap \mathbf{H}^{\circ}$. Hence this is the class of Boolean functions φ that are both model-meet and model-join closed. For all valuations μ and μ', $\mu \cap \mu' \models \varphi$ and $\mu \cup \mu' \models \varphi$ when $\mu \models \varphi$ and $\mu' \models \varphi$. Since $\mathbf{H}$ and $\mathbf{H}^{\circ}$ commute as closure operators, we could equally well have defined $\mathbf{V}_{\rightarrow} = \mathbf{H} \circ \mathbf{H}^{\circ}$.

Syntactically, $\mathbf{V}_{\rightarrow}$ consists of exactly those Boolean functions that can be written in conjunctive normal form $\bigwedge c$ with each clause c taking one of four forms: *0*, x, $\overline{x}$, or $x \rightarrow y$. Note that $\mathbf{V}_{\rightarrow}^{\circ} = \mathbf{V}_{\rightarrow}$.

Lemma 3. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$. Then

$$\text{(a)}\ \ x \vee \varphi \in \mathbf{V}_{\rightarrow} \text{ iff } \varphi \in \mathbf{V}^{\circ} \qquad \text{(b)}\ \ \overline{x} \vee \varphi \in \mathbf{V}_{\rightarrow} \text{ iff } \varphi \in \mathbf{V}$$

Proof: (a) Since $x \vee \varphi \in \mathbf{V}_{\rightarrow}$, we know that $x \vee \varphi \in \mathbf{H}$ and $x \vee \varphi \in \mathbf{H}^{\circ}$. Then by Lemma 2, $\varphi \in \mathbf{M}^{\circ}$ and $\varphi \in \mathbf{H}^{\circ}$. Thus $\varphi \in \mathbf{V}^{\circ}$. The proof for (b) is similar. ∎

3.4 The Classes V, V°, C, and 1

We define $\mathbf{V}$ to be the class of model-meet and upward closed Boolean functions. Syntactically, $\varphi \in \mathbf{V}$ iff $\varphi = 0$ or φ can be written as a (possibly empty) conjunction of positive literals. Dually, $\mathbf{V}^{\circ}$ is the class of model-join and downward closed Boolean functions — those that can be written as *0* or (possibly empty) conjunctions of *negative* literals. $\mathbf{C}$ is the set of Boolean functions that are both upward and downward closed. This set contains only the constant functions *0* and *1*. Finally, $\mathbf{1}$ consists of only the constant function *1*. The next lemma is trivial, but included for completeness.

Lemma 4. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$. Then

(a)	$x \vee \varphi \in \mathbf{V}$ iff $\varphi \in \mathbf{C}$	(e)	$x \vee \varphi \in \mathbf{C}$ iff $\varphi \in \mathbf{1}$
(b)	$\overline{x} \vee \varphi \in \mathbf{V}$ iff $\varphi \in \mathbf{1}$	(f)	$\overline{x} \vee \varphi \in \mathbf{C}$ iff $\varphi \in \mathbf{1}$
(c)	$x \vee \varphi \in \mathbf{V}^{\circ}$ iff $\varphi \in \mathbf{1}$	(g)	$x \vee \varphi \in \mathbf{1}$ iff $\varphi \in \mathbf{1}$
(d)	$\overline{x} \vee \varphi \in \mathbf{V}^{\circ}$ iff $\varphi \in \mathbf{C}$	(h)	$\overline{x} \vee \varphi \in \mathbf{1}$ iff $\varphi \in \mathbf{1}$

Proof: The proof is similar to that of Lemma 3. ∎

4 Algorithms for Approximating ROBDDs

In this section we show how to find upper approximations within the classes of the previous section. Algorithms for lower approximation can be obtained in a parallel way. We assume input and output given as ROBDDs. In this context, the main obstacle to the development of algorithms is a lack of distributivity and substitutivity properties amongst closure operators. To exemplify the problems in the context of $\mathbf{H}$, given $\varphi = x \leftrightarrow y$ and $\psi = x \vee y$, we have

$$\begin{array}{rcl} \mathbf{H}(\varphi \wedge \psi) = \mathbf{H}(x \wedge y) = x \wedge y & \neq & (x \leftrightarrow y) \wedge \mathit{1} = \mathbf{H}(\varphi) \wedge \mathbf{H}(\psi) \\ \mathbf{H}(x \vee y) = \mathit{1} & \neq & x \vee y = \mathbf{H}(x) \vee \mathbf{H}(y) \\ (\mathbf{H}(x \vee y))^{\mathit{0}}_{x} = \mathit{1}^{\mathit{0}}_{x} = \mathit{1} & \neq & y = \mathbf{H}(y) = \mathbf{H}((x \vee y)^{\mathit{0}}_{x}) \end{array}$$

Nevertheless, for a large number of commonly used classes, Boolean development gives us a handle to restore a limited form of distributivity. The idea is as follows. Let $\sigma = (x \wedge \varphi) \vee (\overline{x} \wedge \psi)$. We can write σ alternatively as

$$\sigma = (\overline{\psi} \rightarrow x) \wedge (x \rightarrow \varphi)$$

showing how the "subtrees" φ and ψ communicate with x. As we have seen, we cannot in general find $\rho(\sigma)$, even for "well-behaved" closure operators ρ, by distribution—the following does *not* hold:

$$\rho(\sigma) = \rho(\overline{\psi} \rightarrow x) \wedge \rho(x \rightarrow \varphi)$$

Suppose however that we add a redundant conjunct to the expression for σ:

$$\sigma = (\overline{\psi} \rightarrow x) \wedge (x \rightarrow \varphi) \wedge (\varphi \vee \psi)$$

The term $\varphi \vee \psi$ is redundant, as it is nothing but $\exists x(\sigma)$. The point that we utilise here is that, for a large number of natural classes (or upper closure operators) ρ, distribution is allowed in this context, that is,

$$\rho(\sigma) = \rho(\overline{\psi} \rightarrow x) \wedge \rho(x \rightarrow \varphi) \wedge \rho(\varphi \vee \psi)$$

The intuition is that the information that is lost by $\rho(\overline{\psi} \rightarrow x) \wedge \rho(x \rightarrow \varphi)$, namely the "$\rho$" information shared by φ and ψ is exactly recovered by $\rho(\varphi \vee \psi)$. Figure 3 shows, for reference, the ROBDD for a function, $(x \rightarrow z) \wedge (\overline{y} \rightarrow z)$, and the ROBDDs that result from different approximations.

Before we present our approximation algorithms, we need one more lemma.

Lemma 5. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$.

(a) **if** $x \vee \varphi \in \Delta \leftrightarrow \varphi \in \Delta'$ **then** $\Delta(x \vee \varphi) = x \vee \Delta'(\varphi)$
(b) **if** $\overline{x} \vee \varphi \in \Delta \leftrightarrow \varphi \in \Delta'$ **then** $\Delta(\overline{x} \vee \varphi) = \overline{x} \vee \Delta'(\varphi)$

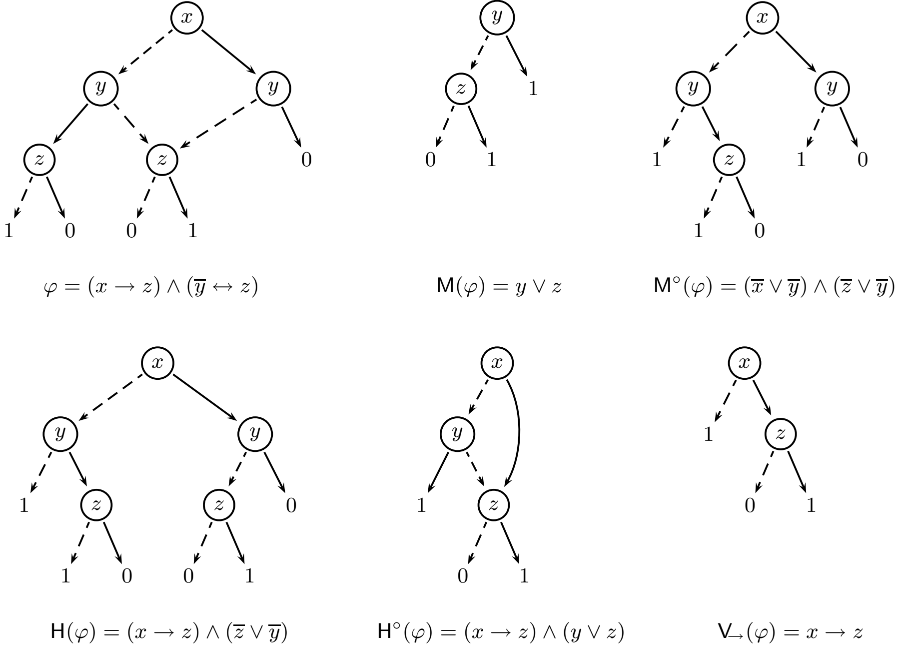


Fig. 3. ROBDDs for $(x \to z) \wedge (\overline{y} \leftrightarrow z)$ and some approximations of it

Proof: We show (a)—the proof for (b) is similar. Assume $x \vee \varphi \in \Delta \leftrightarrow \varphi \in \Delta'$.

$$
\begin{aligned}
\Delta(x \vee \varphi) &= \textstyle\bigwedge\{\psi' \in \Delta \mid x \vee \varphi \models \psi'\} \\
&= \textstyle\bigwedge\{\psi' \in \Delta \mid x \models \psi' \text{ and } \varphi \models \psi'\} \\
&= \textstyle\bigwedge\{x \vee \psi \mid x \vee \psi \in \Delta \text{ and } \varphi \models x \vee \psi\} && \psi' \text{ is of the form } x \vee \psi \\
&= \textstyle\bigwedge\{x \vee \psi \mid x \vee \psi \in \Delta \text{ and } \varphi \models \psi\} && \varphi \text{ is independent of } x \\
&= \textstyle\bigwedge\{x \vee \psi \mid \psi \in \Delta' \text{ and } \varphi \models \psi\} && \text{premise} \\
&= x \vee \textstyle\bigwedge\{\psi \mid \psi \in \Delta' \text{ and } \varphi \models \psi\} \\
&= x \vee \Delta'(\varphi).
\end{aligned}
$$

∎

4.1 The Upper Closure Operators H and H°

Algorithm 1. To find the strongest **H** consequence of a Boolean function:

$$
\begin{aligned}
\mathsf{H}(0) &= 0 \\
\mathsf{H}(1) &= 1 \\
\mathsf{H}(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, R^t, R^f) \\
&\quad \textbf{where } R' = \mathsf{H}(\mathsf{or}(R_1, R_2)) \\
&\quad \textbf{and } R^t = \mathsf{H}(R_1) \\
&\quad \textbf{and } R^f = \mathsf{and}(\mathsf{M}^\circ(R_2), R')
\end{aligned}
$$

To prove the correctness of this algorithm, we shall need the following lemma:

Lemma 6. Let $\varphi \in \mathbf{B}$ be independent of $x \in \mathcal{V}$. Then

(a) $\mathbf{H}(x \wedge \varphi) = x \wedge \mathbf{H}(\varphi)$ (c) $\mathbf{H}^\circ(x \wedge \varphi) = x \wedge \mathbf{H}^\circ(\varphi)$
(b) $\mathbf{H}(\overline{x} \wedge \varphi) = \overline{x} \wedge \mathbf{H}(\varphi)$ (d) $\mathbf{H}^\circ(\overline{x} \wedge \varphi) = \overline{x} \wedge \mathbf{H}^\circ(\varphi)$ ∎

Proposition 1. For any ROBDD R, $\mathbf{H}[\![R]\!] = [\![\mathsf{H}(R)]\!]$.

Proof: By induction on $\mathsf{vars}(R)$. When $\mathsf{vars}(R) = \varnothing$, R must be either 0 or 1; in these cases the proposition holds.

Assume $\mathsf{vars}(R) \neq \varnothing$ and take $R = \mathsf{ite}(x, R_1, R_2)$. $\mathsf{vars}(R) \supset \mathsf{vars}(\mathsf{or}(R_1, R_2))$, so the induction is well-founded. Let $R' = \mathsf{H}(\mathsf{or}(R_1, R_2))$. By the induction hypothesis, $[\![R']\!] = \mathbf{H}[\![\mathsf{or}(R_1, R_2)]\!] = \mathbf{H}([\![R_1]\!] \vee [\![R_2]\!])$.

We prove first that $\mathbf{H}[\![R]\!] \models [\![\mathsf{H}(R)]\!]$. Note that

$$\begin{array}{lll} x \vee \mathbf{H}(\psi) & \models \mathbf{H}(x \vee \mathbf{H}(\psi)) & = \mathbf{H}(x \vee \psi) \\ \overline{x} \vee \mathbf{H}(\varphi) & \models \mathbf{H}(\overline{x} \vee \mathbf{H}(\varphi)) & = \mathbf{H}(\overline{x} \vee \varphi) \\ \mathbf{H}(\varphi) \vee \mathbf{H}(\psi) & \models \mathbf{H}(\mathbf{H}(\varphi) \vee \mathbf{H}(\psi)) & = \mathbf{H}(\varphi \vee \psi) \end{array}$$

Since $\mathbf{H}$ and $\wedge$ are monotone, $\mathbf{H}[(x \vee \mathbf{H}(\psi)) \wedge (\overline{x} \vee \mathbf{H}(\varphi)) \wedge (\mathbf{H}(\varphi) \vee \mathbf{H}(\psi))] \models \mathbf{H}[\mathbf{H}(x \vee \psi) \wedge \mathbf{H}(\overline{x} \vee \varphi) \wedge \mathbf{H}(\varphi \vee \psi)]$. Hence, by (1),

$$\begin{array}{c} \mathbf{H}[(x \vee \mathbf{H}(\psi)) \wedge (\overline{x} \vee \mathbf{H}(\varphi)) \wedge (\mathbf{H}(\varphi) \vee \mathbf{H}(\psi))] \\ \models \mathbf{H}(x \vee \psi) \wedge \mathbf{H}(\overline{x} \vee \varphi) \wedge \mathbf{H}(\varphi \vee \psi) \end{array} \tag{3}$$

Now we have

$$\begin{array}{ll}
\mathbf{H}[\![R]\!] & \\
= \mathbf{H}[(x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!])] & \\
= \mathbf{H}(\mathbf{H}(x \wedge [\![R_1]\!]) \vee \mathbf{H}(\overline{x} \wedge [\![R_2]\!])) & \text{uco property} \\
= \mathbf{H}[(x \wedge \mathbf{H}[\![R_1]\!]) \vee (\overline{x} \wedge \mathbf{H}[\![R_2]\!])] & \text{Lemma 6} \\
= \mathbf{H}[(x \vee \mathbf{H}[\![R_2]\!]) \wedge (\overline{x} \vee \mathbf{H}[\![R_1]\!]) & \text{distribution} \\
= \mathbf{H}[(x \vee \mathbf{H}[\![R_2]\!]) \wedge (\overline{x} \vee \mathbf{H}[\![R_1]\!]) \wedge (\mathbf{H}[\![R_1]\!] \vee \mathbf{H}[\![R_2]\!])] & \\
\models \mathbf{H}(x \vee [\![R_2]\!]) \wedge \mathbf{H}(\overline{x} \vee [\![R_1]\!])) \wedge \mathbf{H}([\![R_1]\!] \vee [\![R_2]\!]) & \text{Equation 3} \\
= (x \vee \mathbf{M}^\circ[\![R_2]\!]) \wedge (\overline{x} \vee \mathbf{H}[\![R_1]\!]) \wedge [\![R']\!] & \text{Lemmas 2 and 5} \\
= (x \wedge \mathbf{H}[\![R_1]\!] \wedge [\![R']\!]) \vee (\overline{x} \wedge \mathbf{M}^\circ[\![R_2]\!] \wedge [\![R']\!]) & \text{distribution} \\
= (x \wedge \mathbf{H}[\![R_1]\!]) \vee (\overline{x} \wedge \mathbf{M}^\circ[\![R_2]\!] \wedge [\![R']\!]) & \mathbf{H} \text{ is monotone} \\
= (x \wedge [\![\mathsf{H}(R_1)]\!]) \vee (\overline{x} \wedge [\![\mathsf{M}^\circ(R_2)]\!] \wedge [\![R']\!]) & \text{Ind. hyp., Prop 4} \\
= [\![\mathsf{mknd}(x, \mathsf{H}(R_1), \mathsf{and}(\mathsf{M}^\circ(R_2), R'))]\!] & \\
= [\![\mathsf{H}(R)]\!] &
\end{array}$$

Next we show $[\![\mathsf{H}(R)]\!] \models \mathbf{H}[\![R]\!]$. From the development above, it is clear that this amounts to showing that

$$(x \wedge \mathbf{H}[\![R_1]\!] \wedge [\![R']\!]) \vee (\overline{x} \wedge \mathbf{M}^\circ[\![R_2]\!] \wedge [\![R']\!]) \models \mathbf{H}[(x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!])]$$

By Lemma 6, $x \wedge \mathbf{H}[\![R_1]\!] = \mathbf{H}(x \wedge [\![R_1]\!])$. So clearly $x \wedge \mathbf{H}[\![R_1]\!] \wedge [\![R']\!] \models \mathbf{H}(x \wedge [\![R_1]\!]) \vee \mathbf{H}(\overline{x} \wedge [\![R_2]\!])$, so $x \wedge \mathbf{H}[\![R_1]\!] \wedge [\![R']\!] \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. It remains to prove that $\overline{x} \wedge \mathbf{M}^\circ[\![R_2]\!] \wedge [\![R']\!] \models \mathbf{H}[(x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!])]$. If the left-hand side

is false, then the claim holds trivially. So let μ be a model of $\overline{x} \wedge \mathbf{M}^\circ[\![R_2]\!] \wedge [\![R']\!]$. Thus $\mu \models \overline{x}$, $\mu \models \mathbf{M}^\circ[\![R_2]\!]$, and $\mu \models [\![R']\!]$, and we must show μ entails the right-hand side. Let us consider three exhaustive cases.

First assume $\mu \models \mathbf{H}[\![R_2]\!]$. Then since $\mu \models \overline{x}$, and by Lemma 6, $\mu \models \mathbf{H}(\overline{x} \wedge [\![R_2]\!])$, so certainly $\mu \models \mathbf{H}(x \wedge [\![R_1]\!]) \vee \mathbf{H}(\overline{x} \wedge [\![R_2]\!])$. Then μ must entail the weaker $\mathbf{H}(\mathbf{H}(x \wedge [\![R_1]\!]) \vee \mathbf{H}(\overline{x} \wedge [\![R_2]\!]))$, which by uco properties is equivalent to $\mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$.

Next assume $\mu \models \mathbf{H}[\![R_1]\!]$ and $\mu \not\models \mathbf{H}[\![R_2]\!]$. Since $\mu \models \mathbf{M}^\circ[\![R_2]\!]$, we know that there is some μ' such that $\mu' \models [\![R_2]\!]$, and that $\mu \subseteq \mu'$. Then $\mu' \setminus \{x\} \models \overline{x} \wedge \mathbf{H}[\![R_2]\!] \vee (x \wedge [\![R_1]\!])$, so $\mu' \setminus \{x\}$ must entail the weaker $\mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. We have also assumed $\mu \models \mathbf{H}[\![R_1]\!]$, so by similar argument $\mu \cup \{x\} \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. Then $(\mu \cup \{x\}) \cap (\mu' \setminus \{x\}) \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. But since $\mu \subseteq \mu'$, and since $x \notin \mu$, $(\mu \cup \{x\}) \cap (\mu' \setminus \{x\}) = \mu$, and therefore $\mu \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$.

Finally, assume $\mu \not\models \mathbf{H}[\![R_1]\!]$ and $\mu \not\models \mathbf{H}[\![R_2]\!]$. Since $\mu \models [\![R']\!]$, μ must be the intersection models of $\mathbf{H}[\![R_1]\!]$ and $\mathbf{H}[\![R_2]\!]$. So let μ^+ and μ^- be interpretations such that $\mu^+ \models \mathbf{H}[\![R_1]\!]$ and $\mu^- \models \mathbf{H}[\![R_2]\!]$ and $\mu = \mu^+ \cap \mu^-$. Then, similar to the previous case, $(\mu^+ \cup \{x\}) \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$ and $(\mu^- \setminus \{x\}) \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. But since $\mu = \mu^+ \cap \mu^-$, we know $(\mu^+ \cup \{x\}) \cap (\mu^- \setminus \{x\}) = \mu$, and therefore $\mu \models \mathbf{H}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!]))$. ∎

Algorithm 2. To find the strongest $\mathbf{H}^\circ$ consequence of a Boolean function:

$$
\begin{aligned}
\mathsf{H}^\circ(0) &= 0\\
\mathsf{H}^\circ(1) &= 1\\
\mathsf{H}^\circ(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, R^t, R^f)\\
&\quad \textbf{where } R' = \mathsf{H}^\circ(\mathsf{or}(R_1, R_2))\\
&\quad\quad \textbf{and } R^t = \mathsf{and}(\mathsf{M}(R_1), R')\\
&\quad\quad \textbf{and } R^f = \mathsf{H}^\circ(R_2)
\end{aligned}
$$

Proposition 2. For any ROBDD R, $\mathbf{H}^\circ[\![R]\!] = [\![\mathsf{H}^\circ(R)]\!]$.

Proof: Similar to Proposition 1. ∎

4.2 The Upper Closure Operators M and M°

The algorithms and proofs for $\mathbf{M}$ and $\mathbf{M}^\circ$ are simpler, because these closure operators are additive.

Algorithm 3. To find the strongest $\mathbf{M}$ consequence of a Boolean function:

$$
\begin{aligned}
\mathsf{M}(0) &= 0\\
\mathsf{M}(1) &= 1\\
\mathsf{M}(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, \mathsf{or}(R_1', R_2'), R_2')\\
&\quad \textbf{where } R_1' = \mathsf{M}(R_1)\\
&\quad\quad \textbf{and } R_2' = \mathsf{M}(R_2)
\end{aligned}
$$

Proposition 3. For any ROBDD R, $\mathbf{M}[\![R]\!] = [\![\mathsf{M}(R)]\!]$.

Proof: By structural induction. For $R = 0$ and $R = 1$ the proposition clearly holds. Consider $R = \mathsf{ite}(x, R_1, R_2)$ and let $R_1' = \mathsf{M}(R_1)$ and $R_2' = \mathsf{M}(R_2)$.

$$\begin{aligned}
\mathbf{M}[\![R]\!] &= \mathbf{M}((x \wedge [\![R_1]\!]) \vee (\overline{x} \wedge [\![R_2]\!])) \\
&= (\mathbf{M}(x) \wedge \mathbf{M}[\![R_1]\!]) \vee (\mathbf{M}(\overline{x}) \wedge \mathbf{M}[\![R_2]\!]) && \mathbf{M} \text{ is additive} \\
&= (x \wedge \mathbf{M}[\![R_1]\!]) \vee \mathbf{M}[\![R_2]\!] \\
&= (x \wedge [\![R_1']\!]) \vee [\![R_2']\!] && \text{induction hypothesis} \\
&= (x \wedge ([\![R_1']\!] \vee [\![R_2']\!])) \vee (\overline{x} \wedge [\![R_2']\!]) && \text{development around } x \\
&= (x \wedge [\![\mathsf{or}(R_1', R_2')]\!]) \vee (\overline{x} \wedge [\![R_2']\!]) \\
&= [\![\mathsf{mknd}(x, \mathsf{or}(R_1', R_2'), R_2')]\!] \\
&= [\![\mathsf{M}(R)]\!]
\end{aligned}$$ ■

Algorithm 4. To find the strongest $\mathbf{M}^\circ$ consequence of a Boolean function:

$$\begin{aligned}
\mathsf{M}^\circ(0) &= 0 \\
\mathsf{M}^\circ(1) &= 1 \\
\mathsf{M}^\circ(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, R_1', \mathsf{or}(R_1', R_2')) \\
&\quad \textbf{where } R_1' = \mathsf{M}^\circ(R_1) \\
&\quad \textbf{and } R_2' = \mathsf{M}^\circ(R_2)
\end{aligned}$$

Proposition 4. For any ROBDD R, $\mathbf{M}^\circ[\![R]\!] = [\![\mathsf{M}^\circ(R)]\!]$.

Proof: Similar to Proposition 3. ■

4.3 The Upper Closure Operator $\mathbf{V}_{\rightarrow}$

Algorithm 5. To find the strongest $\mathbf{V}_{\rightarrow}$ consequence of a Boolean function:

$$\begin{aligned}
\mathsf{V}_{\rightarrow}(0) &= 0 \\
\mathsf{V}_{\rightarrow}(1) &= 1 \\
\mathsf{V}_{\rightarrow}(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, \mathsf{and}(\mathsf{V}(R_1), R'), \mathsf{and}(\mathsf{V}^\circ(R_2), R')) \\
&\quad \textbf{where } R' = \mathsf{V}_{\rightarrow}(\mathsf{or}(R_1, R_2))
\end{aligned}$$

Proposition 5. For any ROBDD R, $\mathbf{V}_{\rightarrow}[\![R]\!] = [\![\mathsf{V}_{\rightarrow}(R)]\!]$.

Proof: This follows from the fact that $\mathbf{V}_{\rightarrow} = \mathbf{H} \cap \mathbf{H}^\circ$. We omit the details. ■

4.4 The Upper Closure Operators C, V, and $\mathbf{V}^\circ$

The remaining algorithms are given here for completeness. Their correctness proofs are straightforward.

Algorithm 6. To find the strongest $\mathbf{V}$ consequence of a Boolean function:

$$\begin{aligned}
\mathsf{V}(0) &= 0 & \mathsf{V}(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, R', \mathsf{and}(\mathsf{C}(R_2), R')) \\
\mathsf{V}(1) &= 1 & &\quad \textbf{where } R' = \mathsf{V}(\mathsf{or}(R_1, R_2))
\end{aligned}$$

Algorithm 7. To find the strongest $\mathbf{V}^\circ$ consequence of a Boolean function:

$$\begin{aligned}
\mathsf{V}^\circ(0) &= 0 & \mathsf{V}^\circ(\mathsf{ite}(x, R_1, R_2)) &= \mathsf{mknd}(x, \mathsf{and}(\mathsf{C}(R_1), R'), R') \\
\mathsf{V}^\circ(1) &= 1 & &\quad \textbf{where } R' = \mathsf{V}^\circ(\mathsf{or}(R_1, R_2))
\end{aligned}$$

Algorithm 8. To find the strongest $\mathbf{C}$ consequence of a Boolean function:

$$\mathsf{C}(0) = 0 \qquad \mathsf{C}(1) = 1 \qquad \mathsf{C}(\mathsf{ite}(x, R_1, R_2)) = 1$$

5 Discussion and Related Work

The classes we have covered are but a few examples of the generality of our approach. Many other classes fall under the same general scheme as the algorithms in Section 4. One such is **L**. Syntactically, $\varphi \in \mathbf{L}$ iff $\varphi = \mathit{0}$ or φ can be written as a (possibly empty) conjunction of literals. Slightly more general is the class **Bij** of *bijunctive* functions. Members of this class can be written in clausal form with at most two literals per clause.

A class central to many analyses of logic programs is that of *positive* functions [16, 17]. Let $\mu_\top$ be the unit valuation, that is, $\mu_\top = \mathit{1}$ for all $x \in \mathcal{V}$. Then φ is positive iff $\mu_\top \models \varphi$. We denote the class of positive functions by **Pos**. This class is interesting in the context of ROBDDs, as it is a class which is easily recognisable but problematic to find approximations in. To decide whether an ROBDD represents a positive function, simply follow the solid-arc path from the root to a sink—the function is positive if and only if the sink is *1*. Approximation, however, can not be done in general without knowledge of the entire space of variables, and not all variables necessarily appear in the ROBDD. For example, if the set of variables is $\mathcal{V}$, then $\mathbf{Pos}(\overline{x_i}) = x_i \rightarrow \bigwedge \mathcal{V}$, which depends on every variable in $\mathcal{V}$. We should note, however, that this does not mean that our approximation algorithms are useless for sub-classes of **Pos**. On the contrary, they work seamlessly for the positive sub-classes commonly used in program analysis, discussed below, as long as positive functions are being approximated (which is invariably the case).

The classes we have discussed above are not sub-classes of **Pos**. (In both **M** and **V**, however, the only non-positive element is *0*.) Restricting the classes to their positive counterparts, we obtain classes that all have found use in program analysis. Figure 4 shows the correspondence. The classes on the right are obtained by intersecting those on the left with **Pos**. We mention just a few example uses. In the context of groundness analysis for constraint logic programs, **Pos** and **Def** are discussed by Armstrong *et al.* [1]. **Def** is used for example by Howe and King [15]. **2IMP** is found in the exception analysis of Glynn *et al.* [12]. We have also omitted characterizations and algorithms for $\mathbf{V}_\leftrightarrow$, the class of functions that can be written as conjunctions of literals and biimplications of the form $x \leftrightarrow y$ with $x, y \in \mathcal{V}$. This class corresponds to the set of all possible partitionings of $\mathcal{V} \cup \{\mathit{0}, \mathit{1}\}$. Its restriction to the positive fragment is exactly Heaton *et al.*'s "*EPos*" domain [14].

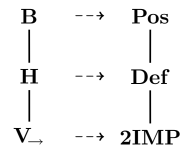


Fig. 4. Positive fragments

M is a class which is of considerable interest in many contexts. In program analysis it has a classical role: Mycroft's well-known two-valued strictness analysis for first-order functional programs [18] uses **M** to capture non-termination information.

The classes we have considered are of much theoretical interest. The classes **Bij**, **H**, $\mathbf{H}^\circ$, **Pos** and $\mathbf{Pos}^\circ$ are five of the six classes from Schaefer's dichotomy result [22] (the sixth is the class of affine Boolean functions). **M** plays a role in

Post's functional completeness result [20], together with the affine functions, **Pos** and its dual, and the class of self-dual functions. Giacobazzi and Scozzari provide interesting characterisations of domains including **Pos** in terms of domain completion using natural domain operations [11].

The problem of *approximating* Boolean functions appears in many contexts in program analysis. We already mentioned Genaim and King's suspension analysis [9] and the formulation of set-sharing using **Pos**, by Codish *et al.* [5]. Another possible application is in the design of widening operators for abstract interpretation-based analyses.

6 Conclusion

We have provided algorithms to find upper approximations for Boolean functions represented as ROBDDs. The algorithms all follow the same general pattern, which works for a large number of important classes of Boolean functions. They also provide a way of checking an ROBDD R for membership of a given class $\varDelta$: Simply check whether $R = \varDelta(R)$.

In the design of our algorithms we have emphasised clarity rather than efficiency. We note that the critical term $\varDelta(\varphi \vee \psi)$ is identical to the join $\varDelta(\varphi) \sqcup_{\varDelta} \varDelta(\psi)$, so in many cases, efficient approximation algorithms may boil down to efficient computation of the join. Future research will include a search for appropriate data structures and associated complexity analyses, as well as attempts at a more general and abstract approach to the algorithms and proofs.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. Twenty-seventh ACM/IEEE Design Automation Conf.*, pages 40–45, 1990.
3. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, C–35(8):677–691, 1986.
4. R. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
5. M. Codish, H. Søndergaard, and P. J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.
6. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1978.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. Sixth ACM Symp. Principles of Programming Languages*, pages 269–282. ACM Press, 1979.
8. O. Ekin, S. Foldes, P. L. Hammer, and L. Hellerstein. Equational characterizations of Boolean function classes. *Discrete Mathematics*, 211:27–51, 2000.

9. S. Genaim and A. King. Goal-independent suspension analysis for logic programs with dynamic scheduling. In P. Degano, editor, *Proc. European Symp. Programming 2006*, volume 2618 of *LNCS*, pages 84–98. Springer, 2003.
10. R. Giacobazzi. *Semantic Aspects of Logic Program Analysis.* PhD thesis, University of Pisa, Italy, 1993.
11. R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Trans. Programming Languages and Systems*, 20(5):1067–1109, 1998.
12. K. Glynn, P. J. Stuckey, M. Sulzmann, and H. Søndergaard. Exception analysis for non-strict languages. In *Proc. 2002 ACM SIGPLAN Int. Conf. Functional Programming*, pages 98–109. ACM Press, 2002.
13. P. R. Halmos. *Lectures on Boolean Algebras.* Springer-Verlag, 1963.
14. A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A simple polynomial groundness analysis for logic programs. *J. Logic Programming*, 45(1-3):143–156, 2000.
15. J. M. Howe and A. King. Efficient groundness analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.
16. J. M. Howe, A. King, and L. Lu. Analysing logic programs by reasoning backwards. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 152–188. Springer, 2004.
17. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Programming Languages and Systems*, 2(1–4):181–196, 1993.
18. A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, Scotland, 1981.
19. O. Ore. Combinations of closure relations. *Ann. Math.*, 44(3):514–533, 1943.
20. E. L. Post. *The Two-Valued Iterative Systems of Mathematical Logic.* Princeton University Press, 1941. Reprinted in M. Davis, Solvability, Provability, Definability: The Collected Works of Emil L. Post, pages 249–374, Birkhaüser, 1994.
21. S. Rudeanu. *Boolean Functions and Equations.* North-Holland, 1974.
22. T. J. Schaefer. The complexity of satisfiability problems. In *Proc. Tenth Ann. ACM Symp. Theory of Computing*, pages 216–226, 1978.
23. M. Ward. The closure operators of a lattice. *Ann. Math.*, 43(2):191–196, 1942.

A CLP Method for Compositional and Intermittent Predicate Abstraction

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing, National University of Singapore,
S16, 3 Science Drive 2, Singapore 117543,
Republic of Singapore
{joxan, andrews, razvan}@comp.nus.edu.sg

Abstract. We present an implementation of symbolic reachability analysis with the features of compositionality, and *intermittent* abstraction, in the sense of pefroming approximation only at selected program points, if at all. The key advantages of compositionality are well known, while those of intermittent abstraction are that the abstract domain required to ensure convergence of the algorithm can be minimized, and that the cost of performing abstractions, now being intermittent, is reduced.

We start by formulating the problem in CLP, and first obtain compositionality. We then address two key efficiency challenges. The first is that reasoning is required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This essentially means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. This is addressed by using the variable elimination or projection mechanism that is implicit in CLP systems. The second challenge is termination, that is, to determine which subgoals are redundant. We address this by a novel formulation of memoization called *coinductive tabling*.

We finally evaluate the method experimentally. At one extreme, where abstraction is performed at every step, we compare against a model checker. At the other extreme, where no abstraction is performed, we compare against a program verifier. Of course, our method provides for the middle ground, with a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants.

1 Introduction

Predicate abstraction [15] is a successful method of abstract interpretation. The abstract domain, constructed from a given finite set of predicates over program variables, is intuitive and easily, though not necessarily efficiently, computable within a traversal method of the program's control flow structure.

While it is generally straightforward to optimize the process of abstraction to a certain extent by performing abstraction at selected points only (eg. several consecutive asignments may be compressed and abstraction performed accross one composite assignment, as implemented in the BLAST system [19]), to this point there has not been a systematic way of doing this. Moreover, since the abstract description is limited to a fixed number of variables, such an ad-hoc method would not be compositional. For example, [2] requires an elaborate extension of predicate abstraction which essentially

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 17–32, 2006.
© Springer-Verlag Berlin Heidelberg 2006

```
⟨0⟩ i := 0 ; c := 0
⟨1⟩ while (i < n) do
⟨2⟩  i++
⟨3⟩  c++
⟨4⟩  c++
⟨5⟩ end ⟨6⟩
```

(a)

$even(0,i,n,c) \mapsto even(1,i_1,n_1,c_1), i_1 = 0, n_1 = n, c_1 = 0.$
$even(1,i,n,c) \mapsto even(2,i_1,n_1,c_1), i_1 = i, n_1 = n, i < n, c_1 = c.$
$even(2,i,n,c) \mapsto even(3,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c.$
$even(3,i,n,c) \mapsto even(4,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c+1.$
$even(4,i,n,c) \mapsto even(5,i_1,n_1,c_1), i_1 = i+1, n_1 = n, c_1 = c+1.$
$even(5,i,n,c) \mapsto even(2,i_1,n_1,c_1), i_1 = i, n_1 = n, c_1 = c, i < n.$
$even(5,i,n,c) \mapsto even(6,i_1,n_1,c_1), i_1 = i, n_1 = n, c_1 = c, i \geq n.$

(b)

Fig. 1. Even counts

considers a second set of variables (called "symbolic constants"), in order to describe the behaviour of a *function*, in the language of predicate abstraction. This provides only a limited form of compositionality.

In this paper, we present a way of engineering a general proof method of program reasoning based on predicate abstraction in which the process of abstraction is intermittent, that is, approximation is performed only at selected program points, if at all. There is no restriction of when abstraction is performed, even though termination issues will usually restrict the choices. The key advantages are that (a) the abstract domain required to ensure convergence of the algorithm can be minimized, and (b) the cost of performing abstractions, now being intermittent, is reduced.

For example, to reason that $x = 2$ after executing `x := 0; x++; x++`, one needs to know that $x = 1$ holds before the final assignment. Thus, in a predicate abstraction setting, the abstract domain must contain the predicate $x = 1$ for the above reasoning to be possible. Also, consider proving $x = 2n$ for the program snippet in Figure 1a. A textbook Hoare-style loop invariant for the loop is $c = 2i$. Having this formula in the abstract domain would, however, not suffice; one in fact needs to know that $c = 2i - 1$ holds in between the two increments to c. Thus in general, a proper loop invariant is useful only if we could propagate its information throughout the program *exactly*.

A main challenge with exact propagation is that reasoning will be required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This essentially means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. The advantages in terms of efficiency, however, are significant: less predicates needed in the abstract domain, and also, less frequent execution of the abstraction operation. Alternatively, it may be argued that using the weakest precondition operator for exact propagation may result in a set of constraints over a constant number of variables, and thus circumvent the challenge mentioned above. To see that this is not true, let us consider the following program fragment: `while(x%7!=0)x++ ; while(x%11!=0)x++`. Also, let us assume that we have an exact propagation algorithm, based on either the weakest preconditon or the strongest postcondition propagation operator, which computes a constraint that reflects the relationship between the values of `x` before and after the execution of the program fragment. Our algorithm needs to record the fact that between the two while loops the value of `x` is a multiple of 7. This cannot be done without introducing an auxilliary variable in the set of constraints. Assume now that this program

fragment appears in the body of another loop. Since that (outer) loop may be traversed multiple times in the analysis process, and every traversal of the loop will introduce a new auxilliary variable, the number of auxilliary variables is potentially unbounded, irrespective of the propagation operator that is used.

An important feature of our proof method is that it is compositional. We represent a proof as a Hoare-style triple which, for a given program fragment, relates the input values of the variables to the output values. This is represented as a formula, and in general, such a formula must contain auxiliary variables in addition to the program variables. This is because it is generally impossible to represent the projection of a formula using a predefined set of variables, or equivalently, it is not possible to perform quantifier elimination. Consequently, in order to have unrestricted composition of such proofs, it is (again) necessary to deal with an unbounded number of variables.

The paper is organized as follows. We start by formulating the problem in CLP, and first obtain compositionality. We then address two key efficiency challenges. The first is that reasoning is required about the strongest-postcondition operator associated with an arbitrarily long program fragment. This means dealing with constraints over an unbounded number of variables describing the states between the start and end of the program fragment at hand. We address this problem by using the variable elimination or projection mechanism that is implicit in CLP systems. The second challenge is termination, which translates into determining the redundancy of subgoals. We address this by a novel formulation of memoization called *coinductive tabling*.

We finally evaluate the method experimentally. At one extreme, where abstraction is performed at every step, we compare against the model checker BLAST [19]. Here we employ a standard realization of intermittence by abstracting at prespecified points, and thus our algorithm becomes automatic. At the other extreme, where no abstraction is performed (but where invariants are used to deal with loops), we compare against the program-verifier ESC/Java [6]. Of course, our method provides for the middle ground, with a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants.

In summary, we present a CLP-based proof method which has the properties of being compositional, and which employs intermittent abstraction. The major technical contributions, toward this goal, are: the CLP formulation of the proof obligation, which provides expressiveness, and compositionality; a coinduction principle, which provides the basic mechanism for termination; and engineering the use of the underlying CLP projection mechanism in the process of exact propagation. Our method thus provides a flexible combination of abstraction and Hoare-style reasoning with predicate transformers and loop-invariants, that is compositional, and its practical implementation is feasible.

1.1 Related Work

An important category of tools that use program verification technology have been developed within the framework of the Java Modelling Language (JML) project. JML allows one to specify a Java method's pre- and post-conditions, and class invariants. Examples of such program verification tools are: Jack [11], ESC/Java2 [6], and Krakatoa [24]. All these tools employ weakest precondition/strongest postcondition

calculi to generate proof obligations which reflect whether the given post-conditions and class invariants hold at the end of a method, whenever the corresponding pre-conditions are valid at the procedure's entry point. The resulting proof obligations are subsequently discharged by theorem provers such as Simplify [6], Coq [3], PVS [27], or HOL light [18]. While these systems perform exact propagation, they depend on user-provided loop invariants, as opposed to an abstract domain.

Cousot and Cousot [7] have recognized a long time ago that coarse-grained abstractions are better than fine-grained ones. Moreover, recently there have emerged systems based on abstract interpretation, and in particular, on predicate abstraction. Some examples are BLAST [19], SLAM [1], MAGIC [5], and Murphi– – [8], amongst others. While abstract interpretation is central, these systems employ a further technique of *automatically* determining the abstract domain needed for a given assertion. This technique iteratively refines the abstract domain based on information derived from previous counterexamples. These systems do not perform exact propagation in a systematic way.

The use of CLP for program reasoning is not new (see for example [14] for a non-exhaustive survey). Due to its capability for handling constraints, CLP has been notably used in verification of infinite-state systems [9, 10, 13, 17, 23], although results for finite-state systems are also available [26, 12]. Indeed, it is generally straightforward to represent program transitions as CLP rules, and to use the CLP operational model to prove assertions stated as CLP goals. What is novel in our CLP formulation is firstly, the compositional assertion, and then, coinductive tabling. More importantly, our formulation considers CLP programs, assertions and tabling in full generality.

2 Preliminaries

Apart from a program counter k, whose values are program points, let there be *n system variables* $\tilde{v} = v_1, \cdots, v_n$ with domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$ respectively. In this paper, we shall use just two example domains, that of integers, and that of integer arrays. We assume the number of system variables is larger than the number of variables required by any program fragment or procedure.

Definition 1 (States and Transitions). *A* system state *(or simply state) is of the form* $(k, d_1, \cdots, d_n)$ *where* k *is a program point and* $d_i \in \mathcal{D}_i, 1 \leq i \leq n$*, are values for the system variables. A* transition *is a pair of states.* □

In what follows, we define a language of first-order formulas. Let $\mathcal{V}$ denote an infinite set of variables, each of which has a type in $\mathcal{D}_1, \cdots, \mathcal{D}_n$, let Σ denote a set of *functors*, and Π denote a set of *constraint symbols*. A *term* is either a constant (0-ary functor) in Σ or of the form $f(t_1, \cdots, t_m)$, $m \geq 1$, where $f \in \Sigma$ and each t_i is a term, $1 \leq i \leq m$. A *primitive constraint* is of the form $\phi(t_1, \cdots, t_m)$ where ϕ is a $m-ary$ constraint symbol and each t_i is a term, $1 \leq i \leq m$.

A *constraint* is constructed from primitive constraints using logical connectives in the usual manner. Where Ψ is a constraint, we write $\Psi(\tilde{X})$ to denote that Ψ possibly refers to variables in $\tilde{X}$, and we write $\tilde{\exists}\Psi(\tilde{X})$ to denote the existential closure of $\Psi(\tilde{X})$ over variables distinct from those in $\tilde{X}$.

A *substitution* is a mapping which simultaneously replaces each variable in a term or constraint by some expression. Where e is a term or constraint, we write $e\theta$ to denote

the result of applying θ to e. A *renaming* maps each variable in a given sequence, say $\tilde{X}$, into the corresponding variable in another given sequence, say $\tilde{Y}$. We write $[\tilde{X} \mapsto \tilde{Y}]$ to denote such a mapping. A *grounding substitution*, or simply *grounding* maps each variable of an expression into a ground term representing a value in its respective domain. We denote by $[\![e]\!]$ the set of *all possible* groundings of e.

3 Constraint Transition Systems

A key concept is that a program fragment P operates on a sequence of *anonymous* variables, each corresponding to a system variable at various points in the computation of P. In particular, we consider two sequences $\tilde{x} = x_1, \cdots, x_n$ and $\tilde{x}^t = x_1^t, \cdots, x_n^t$ of anonymous variables to denote the system values before executing P and at the "target" point(s) of P, respectively. Typically, but not always, the target point is the terminal point of P. Our proof obligation or *assertion* is then of the form

$$\{\Psi(\tilde{x})\}\ P\ \{\Psi_1(\tilde{x}, \tilde{x}^t)\}$$

where Ψ and Ψ_1 are constraints over the said variables, and possibly including new variables. Like the Hoare-triple, this states that if P is executed in a state satisfying Ψ, then all states at the target points (if any) satisfy Ψ_1. Note that, unlike the Hoare-triple, P may be nonterminating and Ψ_1 may refer to the states of a point that is reached infinitely often. We will formalize all this below.

For example, let there be just one system variable x, let P be `<0>` x `:=` x `+ 1 <1>`, and let the target point be `<1>`. Then $\{true\}P\{x^t = x+1\}$ holds, meaning P is the successor *function* on x. Similarly, if P were the (perpetual) program `<0> while (`*true*`)` x `:=` x `+ 2 <1> endwhile <2>`, and if `<1>` were the target point, then $\{true\}P\{x^t = x+2z\}$ holds, that is, any state $(1,x)$ at point `<1>` satisfies $\exists z(x^t = x+2z)$. This shows, amongst other things, that the parity of x always remains unchanged.

Our proof method accomodates concurrent programs of a fixed number of processes. Where we have n processes, we shall use as a program point, a sequence of n program points so that the ith program point is one which comes from the ith process, $1 \le i \le n$.

We next represent the program fragment P as a transition system which can be executed symbolically. The following key definition serves two main purposes. First, it is a high level representation of the operational semantics of P, and in fact, it represents its exact *trace* semantics. Second, it is an *executable specification* against which an assertion can be checked.

Definition 2 (Constraint Transition System). *A* constraint transition *of P is a formula*

$$p(k, \tilde{x}) \mapsto p(k_1, \tilde{x_1}), \Psi(\tilde{x}, \tilde{x_1})$$

where k and k_1 are variables over program points, each of $\tilde{x}$ and $\tilde{x_1}$ is a sequence of variables representing a system state, and Ψ is a constraint over $\tilde{x}$ and $\tilde{x_1}$, and possibly some additional auxiliary variables.

A constraint transition system *(CTS) of P is a finite set of constraint transitions of P. The symbol p is called the CTS predicate of P.* □

In what follows, unless otherwise stated, we shall consistently denote by P the program of interest, and by p its CTS predicate.

```
Process 1:                          Process 2:
  while (true) do                     while (true) do
⟨0⟩  x := y + 1                     ⟨0⟩  y := x + 1
⟨1⟩  await (x<y ∨ y=0)              ⟨1⟩  await (y<x ∨ x=0)
⟨2⟩  x := 0                         ⟨2⟩  y := 0
  end                                 end
```

Fig. 2. Two Process Bakery

$bak(0,p2,x,y) \mapsto bak(1,p2,x_1,y), x_1 = y+1.$
$bak(1,p2,x,y) \mapsto bak(2,p2,x,y), x < y \vee y = 0.$
$bak(2,p2,x,y) \mapsto bak(0,p2,x_1,y), x_1 = 0.$
$bak(p1,0,x,y) \mapsto bak(p1,1,x,y_1), y_1 = x+1.$
$bak(p1,1,x,y) \mapsto bak(p1,2,x,y), y < x \vee x = 0.$
$bak(p1,2,x,y) \mapsto bak(p1,0,x,y_1), y_1 = 0.$

Fig. 3. CTS of Two Process Bakery

Consider for example the program in Figure 1a; call it *Even*. Figure 1b shows a CTS for *Even*, whose CTS predicate is *even*.

Consider another example: the Bakery algorithm with two processes in Figure 2. A CTS for this program, call it *Bak*, is given in Figure 3. Note that we use the first and second arguments of the term *bak* to denote the program points of the first and second process respectively.

Clearly the variables in a constraint transition may be renamed freely because their scope is local to the transition. We thus say that a constraint transition is a *variant* of another if one is identical to the other when a renaming subsitution is performed. Further, we may *simplify* a constraint transition by renaming any one of its variables x by an expression y provided that $x = y$ in all groundings of the constraint transition. For example, we may simply state the last constraint transition in Figure 3 into

$$bak(p1,2,x,y) \mapsto bak(p1,0,x,0)$$

by replacing the variable y_1 in the original transition with 0.

The above formulation of program transitions is familiar in the literature for the purpose of defining a set of transitions. What is new, however, is how we use a CTS to define *symbolic* transition sequences, and thereon, the notion of a proof.

By similarity with logic programming, we use the term *goal* to denote a formula that can be subjected to an *unfolding process* in order to infer a logical consequence.

Definition 3 (Goal). *A query or* goal *of a CTS is of the form* $p(k,\tilde{x}), \Psi(\tilde{x})$ *, where k is a program point,* $\tilde{x}$ *is a sequence of variables over system states, and* Ψ *is a constraint over some or all of the variables* $\tilde{x}$*, and possibly some additional variables. The variables* $\tilde{x}$ *are called the* primary *variables of this goal, while any additional variable in* Ψ *is called an* auxiliary *variable of the goal.* □

Thus a goal is just like the conclusion of a constraint transition. We say the goal is a *start goal* if k is the start program point. Similarly, a goal is a *target goal* if k is the target program point. Running a start goal is tantamount to asking the question: which

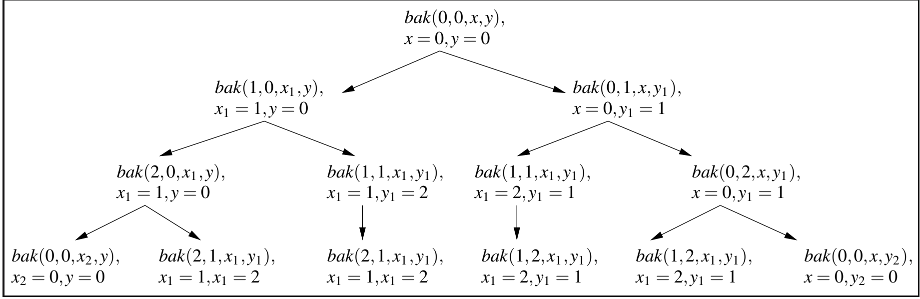


Fig. 4. Proof Tree of 2-Process Bakery Algorithm (Partially Shown)

values of $\tilde{x}$ which satisfy $\tilde{\exists}\Psi(\tilde{x})$ will lead to a goal at the target point(s)? The idea is that we successively reduce one goal to another until the resulting goal is at a target point, and then inspect the results.

Next we define the meaning of proving a goal against a CTS.

Definition 4 (Proof Step, Sequence and Tree). *Let there be a CTS for p, and let $\mathcal{G} = p(k,\tilde{x}),\Psi$ be a goal for this. A* proof step *from $\mathcal{G}$ is obtained via a variant $p(k,\tilde{y}) \mapsto p(k_1,\tilde{y_1}),\Psi_1$ of a transition in the CTS in which all the variables are fresh. The result is a goal of the form $p(k_1,\tilde{y_1}),\Psi,\tilde{x}=\tilde{y},\Psi_1$, providing that the constraints $\Psi,\tilde{x}=\tilde{y},\Psi_1$ are satisfiable.*

A proof sequence *is a finite or infinite sequence of proof steps. A proof tree is defined from proof sequences in the obvious way. A tree is* complete *if every internal node representing a goal $\mathcal{G}$ is succeeded by nodes representing* every *goal obtainable in a proof step from $\mathcal{G}$.* □

Consider again the CTS in Figure 1b, and we wish to prove $\{n=1\}p\{c=2\}$. There is in fact only one proof sequence from the start goal

$$even(0,i,n,c), n=1, c=0.$$

or equivalently, $even(0,i,1,0)$. This proof sequence is shown in Figure 5, and note that the counter, represented in the last goal by the variable c_2, has the value 2.

Definition 5 (Assertion). *Let p be a program with start variables $\tilde{x}$, and let Ψ be a constraint. Let $\tilde{x}^t$ denote a sequence of variables representing system states not appear-*

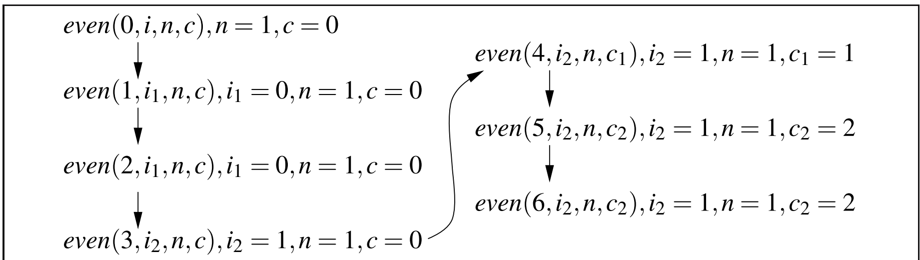


Fig. 5. Proof Tree of Even Counts Program

ing in p or Ψ. (These represent the target values of the system variables.) An assertion for p wrt to $\tilde{x}^t$ is of the form $p(k,\tilde{x}),\Psi \models \Psi_1(\tilde{x},\tilde{x}^t)$. In particular, when k is the start program point, we may abberviate the assertion using the notation $\{\Psi\}p\{\Psi_1\}$. $\square$

It is intuitively clear what it means for an assertion to hold. That is, execution from every instance θ of $p(k,\tilde{x}),\Psi$ cannot lead to a target state where the property $\Psi_1(\tilde{x}\theta,\tilde{x}^t)$ is violated.

In the example above, we could prove the assertion $even(0,i,n,c) \models c^t = 2n$ where it is understood that the final variable c^t corresponds to the start variable c. Note that the last occurrence of n in the assertion means that we are comparing c^t with the initial and not final value of n (though in this example, the two are in fact the same).

We now state the essential property of proof sequences:

Theorem 1. *Let a CTS for p have the start point k and target point k^t, and let $\tilde{x}$ and $\tilde{x_1}$ each be sequences of variables over system states. The assertion $\{\Psi(\tilde{x})\}\ p\ \{\Psi_1(\tilde{x}^t,\tilde{x})\}$ holds if for any goal of the form $p(k^t,\tilde{x_1}),\ \Psi_2(\tilde{x_1},\tilde{x})$ appearing in a proof sequence from the goal $p(k,\tilde{x}),\ \Psi(\tilde{x})$, the following holds: $\tilde{\exists}\Psi_2(\tilde{x_1},\tilde{x}) \models \tilde{\exists}\Psi_1(\tilde{x_1},\tilde{x})$* $\square$

The above theorem provides the basis of a search method, and what remains is to provide a means to ensure termination of the search. Toward this end, we next define the concepts of *subsumption* and *coinduction* and which allow the (successful) termination of proof sequences. However, these are generally insufficient. In the next section, we present our version of *abstraction* whose purpose is to transform a proof sequence so that it is applicable to the termination criteria of subsumption and coinduction.

3.1 Subsumption

Consider a finite and complete proof tree from some start goal. A goal $\mathcal{G}$ in the tree is *subsumed* if there is a different path in the tree containing a goal $\mathcal{G}'$ such that $[\![\mathcal{G}]\!] \subseteq [\![\mathcal{G}']\!]$.

The principle here is simply memoization: one may terminate the expansion of a proof sequence while constructing a proof tree when encountering a subsumed goal.

3.2 Coinduction

The principle here is that, within one proof sequence, the proof obligation associated with the final goal may *assume* that the proof obligation of an ancestor goal has already been met. This can be formally explained as a principle of coinduction (see eg: Appendix B of [25]). Importantly, this simple form of coinduction does not require a base case nor a well-founded ordering.

We shall simply demonstrate this principle by example. Suppose we had the transition $p(0,x) \mapsto p(0,x'),x' = x+2$ and we wished to prove the assertion $p(0,x) \models even(x^t - x)$, that is, the difference between x and its final value is even. Consider the derivation step:

$$p(0,x) \models even(x^t - x)$$
$$p(0,x'),x' = x+2 \models even(x^t - x)$$

We may use, in the latter goal, the fact that the earlier goal satisfies the assertion. That is, we may reduce the obligaton of the latter goal to $even(x^t - x'),x' = x+2 \models even(x^t - x)$.

It is now a simple matter of inferring whether this formula holds. In general practice, the application of coinduction testing is largely equivalent to testing if one goal is simply an instance of another.

4 Abstraction

In the literature on predicate abstraction, the abstract description is a specialized data structure, and the abstraction operation serves to propagate such a structure though a small program fragment (a contiguous group of assignments, or a test), and then obtaining another structure. The strength of this method is in the simplicity of using a finite set of predicates over the fixed number of program variables as a basis for the abstract description.

We choose to follow this method. However, our abstract description shall not be a distinguished data structure. In fact, our abstract description of a goal is itself a goal.

Definition 6 (Abstraction). *An abstraction $\mathcal{A}$ is applied to a goal. It is specified by a program point $pc(\mathcal{A})$, a sequence of variables $var(\mathcal{A})$ corresponding to a subset of the system variables, and finally, a finite set of constraints $pred(\mathcal{A})$ over $var(\mathcal{A})$, called the "predicates" of $\mathcal{A}$.*

Let $\mathcal{A}$ be an abstraction and G be a goal $p(k,\tilde{x}),\Psi$ where $k = pc(\mathcal{A})$. Let $\tilde{x_1}$ denote the subsequence of $\tilde{x}$ corresponding to the system variables $var(\mathcal{A})$. Let $\bar{x}$ denote the remaining subsequence of $\tilde{x}$. Without losing generality, we assume that $\tilde{x_1}$ is an initial subsequence of $\tilde{x}$, that is, $\tilde{x} = \tilde{x_1},\bar{x}$. Then the abstraction $\mathcal{A}(G)$ of G by $\mathcal{A}$ is $p(k,\tilde{Z},\bar{x}),\Psi,\Psi_2[var(\mathcal{A}) \mapsto \tilde{Z}]$, where $\tilde{Z}$ is a sequence of fresh variables renaming $\tilde{x_1}$, and Ψ_2 is the finite set of constraints $\{\psi_2 \in pred(\mathcal{A}) : \Psi \models \psi_2[var(\mathcal{A}) \mapsto \tilde{x_1}]\}$ □

For example, let $\mathcal{A}$ be such that $pc(\mathcal{A}) = 0$, $var(\mathcal{A}) = \{v_1\}$ and $pred(\mathcal{A}) = \{v_1 < 0, v_1 \geq 0\}$. That is, the first variable is to be abstracted into a negative or a nonnegative value. Let G be $p(0,[x_1,x_2,x_3]),x_1 = x_2,x_2 = 1$. Then the abstraction $\mathcal{A}(G)$ is a goal of the form $p(0,[Z,x_2,x_3]),x_1 = x_2,x_2 = 1,Z \geq 0$, which can be simplified into $p(0,[Z,x_2,x_3]),x_2 = 1,Z \geq 0$. Note that the orginal goal had ground instances

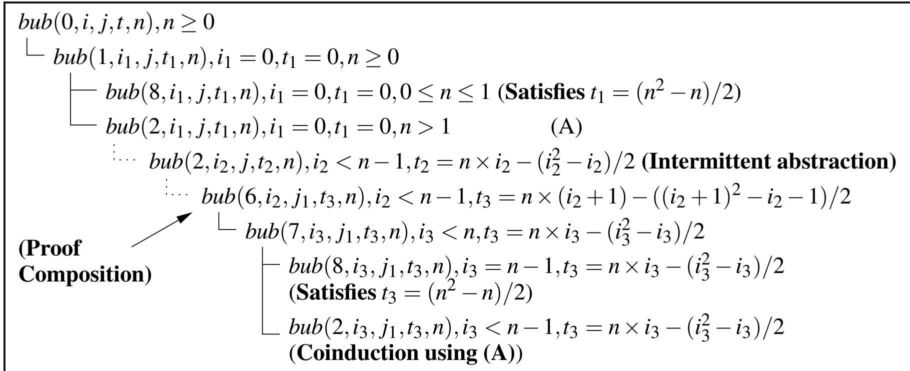


Fig. 6. Compositional Proof

```
⟨0⟩ t := 0; i := 0
⟨1⟩ while (i < n-1) do
⟨2⟩   j := 0
⟨3⟩   while (j < n-i-1) do
⟨4⟩     j := j+1; t := t+1
⟨5⟩   end
⟨6⟩   i := i+1
⟨7⟩ end ⟨8⟩
```

(a)

$bub(0,i,j,t,n) \mapsto bub(1,i_1,j,t_1,n), i_1 = 0, t_1 = 0.$
$bub(1,i,j,t,n) \mapsto bub(8,i,j,t,n), i \geq n-1.$
$bub(1,i,j,t,n) \mapsto bub(2,i,j,t,n), i < n-1.$
$bub(2,i,j,t,n) \mapsto bub(3,i,j_1,t,n), j_1 = 0.$
$bub(3,i,j,t,n) \mapsto bub(6,i,j,t,n), j \geq n-i-1.$
$bub(3,i,j,t,n) \mapsto bub(4,i,j,t,n), j < n-i-1.$
$bub(4,i,j,t,n) \mapsto bub(5,i,j_1,t_1,n), j_1 = j+1, t_1 = t+1.$
$bub(5,i,j,t,n) \mapsto bub(6,i,j,t,n), j \geq n-i-1.$
$bub(5,i,j,t,n) \mapsto bub(4,i,j,t,n), j < n-i-1.$
$bub(6,i,j,t,n) \mapsto bub(7,i_1,j,t_1,n), i_1 = i+1.$
$bub(7,i,j,t,n) \mapsto bub(8,i,j,t,n), i \geq n-1.$
$bub(7,i,j,t,n) \mapsto bub(2,i,j,t,n), i < n-1.$

(b)

Fig. 7. Program "Bubble"

$p(0,[1,1,n])$ for all n, while the abstracted goal has the instances $p(0,[m,1,n])$ for all n and all nonnegative m. Note that the second variable x_2 has not been abstracted even though it is tightly constrained to the first variable x_1. Note further that the value of x_3 is unchanged, that is, the abstraction would allow any constraint on x_3, had the example goal contained such a constraint, to be *propagated*.

Lemma 1. *Let $\mathcal{A}$ be an abstraction and G a goal. Then $[\![G]\!] \subseteq [\![\mathcal{A}(G)]\!]$.* □

The critical point is that the abstraction of a goal has the *same format* as the goal itself. Thus an abstract goal has the expressive power of a regular goal, while yet containing a notion of abstraction that is sufficient to produce a finite-state effect. Once again, this is facilitated by the ability to reason about an unbounded number of variables.

Consider the "Bubble" program and its CTS in Figures 7(a) and 7(b), which is a simplified skeleton of the bubble sort algorithm (without arrays). Consider the subprogram corresponding to start point 2 and whose target point is 6, that is, we are considering the inner loop. Further suppose that the following assertion had already been proven:

$$bub(2,i,j,t,n) \models i^t = i, t^t = t+n-i-1, n^t = n$$

that is, the subprogram increments t by $n-i-1$ while preserving both i and n, but not j. Consider now a proof sequence for the goal $bub(0,i,j,t,n), n \geq 0$, where we want to prove that at program point $\langle 8 \rangle$, $t = (n^2-n)/2$. The proof tree is depicted in Figure 6. The proof shows a combination of the use of intermittent abstraction and compositional proof:

- At point (A), we abstract the goal $bub(2,i_1,j,t_1,n), i_1 = 0, t_1 = 0, n > 1$ using the predicates $i < n-1$ and $t = n \times i - (i^2-i)/2$. Call this abstraction $\mathcal{A}$. Here the set of variables is $var(\mathcal{A}) = \{i,t\}$, hence both the variables i_1 and t_1 that correspond respectively to system variables i and t are renamed to fresh variables i_2, and t_2. Meanwhile, the variables j and n retain their original values.
- After performing the above abstraction, we reuse the proof of the inner loop above. Here we immediately move to program point $\langle 6 \rangle$, incrementing t with $n-i-1$, and updating j to an unknown value. However, i and n retain their original values at $\langle 2 \rangle$.
- The result of the intermittent abstraction above is a coinductive proof.

5 The Whole Algorithm

We now summarize our proof method for an assertion

$$\{\Psi\}p\{\Psi_1\}$$

Suppose the start program point of p is k and the start variables of p are $\tilde{x}$. Then consider the start goal $p(k,\tilde{x}),\Psi$ and incrementally build a complete proof tree. For each path in the tree constructed so far leading to a goal $\mathcal{G}$ if:

- $\mathcal{G}$ is either subsumed or is coinductive, then consider this path *closed*, ie: not to be expanded further;
- $\mathcal{G}$ is a goal on which an abstraction $\mathcal{A}$ is defined, replace $\mathcal{G}$ by $\mathcal{A}(\mathcal{G})$;
- $\mathcal{G}$ is a target goal, and if the constraints on the primary variables $\tilde{x_1}$ in $\mathcal{G}$ do *not* satisfy $\Psi\theta$, where θ renames the target variables in Ψ into $\tilde{x_1}$, terminate and return *false*.
- the expansion of the proof tree is no longer possible, terminate and return *true*.

Theorem 2. *If the above algorithm, applied to the assertion $\{\Psi\}p\{\Psi_1\}$, terminates and does not return false, then the assertion holds.* □

6 CLP Technology

It is almost immediate that CTS is implementable in CLP. Given a CTS for p, we build a CLP program in the following way: (a) for every transition of the form $(k,\tilde{x}) \mapsto (k',\tilde{x'}),\Psi$ we use the CLP rule the clause $\mathtt{p}(k,\tilde{x}) : -\mathtt{p}(k',\tilde{x'}),\Psi$ (assuming that Ψ is in the constraint domain of the CLP implementation at hand); (b) for every terminal program point k, we use the CLP fact $\mathtt{p}(k,_,\ldots,_)$, where the number of anonymous variables is the same as the number of variables in $\tilde{x}$.

We see later that the key implementation challenge for a CLP system is the *incremental satisfiability* problem. Roughly stated, this is the problem of successively determining that a monotonically increasing sequence of constraints (interpreted as a conjunction) is satisfiable.

6.1 Exact Propagation is "CLP-Hard"

Here we informally demonstrate that the incremental satisfiability problem is reducible to the problem of analyzing a straight line path in a program. We will consider here constraints in the form of linear diophantine equations, i.e., multivariate polynomials over the integers. Without loss of generality, we assume each constraint is written in the form $X = Y + Z$ or $X = nY$ where n is an integer. Throughout this section, we denote by X, Y, Z logic variables, and by x, y, z their corresponding program variables, respectively.

Suppose we already have a sequence of constraints $\Psi_0,\cdots,\Psi_i$ and a corresponding path in the program's control flow.

Suppose we add a new constraint $\Psi_{i+1} = (X = Y + Z)$. Then, if one of these variables, say Y, is new, we add the assignment $y := x - z$ where y is a new variable created to correspond to Y. The remaining variables x and z are each either new, or are the corresponding variables to X and Z, respectively. If however all of X, Y and Z are not new,

then add the statement `if (`$x = y+z$`) ...` . Hereafter we pursue the `then` branch of this `if` statement.

Similarly, suppose the new constraint were of the form $X = nY$. Again, if x is new, we simply add the assignment $x := n*y$ where x is newly created to correspond to X. Otherwise, add the statement `if (x = n * y) ...` to the path, and again, we now pursue the `then` branch of this `if` statement.

Clearly an exact analysis of the path we have constructed leading to a successful traversal required, incrementally, the solving of the constraint sequence $\Psi_0, \cdots, \Psi_n$.

6.2 Key Elements of CLP Systems

A CLP system attempts to find answers to an initial goal $\mathcal{G}$ by searching for valid substitutions of its variables, in a depth-first manner. Each path in the search tree in fact involves the solving of an incremental satisfiability problem. Along the way, unsatisfiability of the constraints at hand would entail backtracking.

The key issue in CLP is the incremental satisfiability problem, as mentioned above. A standard approach is as follows. Given that the sequence of constraints $\Psi_0, \ldots, \Psi_i$ has been determined to be satisfiable, represent this fact in a *solved form*. Essentially, this means that when a new constraint Ψ_{i+1} is encountered, the solved form can be combined efficiently with Ψ_{i+1} in order to determine the satisfiability of the new conjunction of constraints.

This method essentially requires a representation of the *projection* of a set of constraints onto certain variables. Consider, for example, the set $x_0 = 0, x_1 = x_1 + 1, x_2 = x_1 + 1, \cdots, x_i = x_{i-1} + 1$. Assuming that the new constraint would only involve the variable x_i (and this happens vastly often), we desire a representation of $x_i = i$. This projection problem is well studied in CLP systems [21]. In the system CLP($\mathcal{R}$) [22] for example, various adaptations of the Fourier-Motzkin algorithm were implemented for projection in Herbrand and linear arithmetic constraints.

We finally mention another important optimization in CLP: *tail recursion*. This technique uses the same space in the procedure call stack for recursive calls. Amongst other benefits, this technique allows for a potentially unbounded number of recursive calls. Tail recursion is particurly relevant in our context because the recursive calls arising from the CTS of programs are often tail-recursive.

The CLP($\mathcal{R}$) system that we use to implement our prototype has been engineered to handle constraints and auxiliary variables efficiently using the above techniques.

7 Experiments

7.1 Exact Runs

We start with an experiment which shows that concrete execution can potentially be less costly than abstract execution. To that end, we compare the timing of concrete execution using our CLP-based implementation and a predicate abstraction-based model checker. We run a simple looping program, whose C code is shown in Figure 8 (a). First, we have BLAST generate all the 100 predicates it requires. We then re-run BLAST by providing these predicates. BLAST took 22.06 seconds to explore the state space.

```
int main()                          int main()
{   int i=0, j, x=0;                {   int i=0, j, x=0;
    while (i<7) {                       while (i<50) {
        j=0;                                i++; j=0;
        while (j<7) { x++; j++; }           while (j<10) { x++; j++; }
        i++; }                              while (x>i) { x--; }}
    if (x>49) { ERROR: }}               if (x<50) { ERROR: }}
```

(a) **(b)**

Fig. 8. Programs with Loop

On the same machine, and without any abstraction, our verification engine took only 0.02 seconds. For comparison, SPIN model checker [20] executes the same program written in PROMELA in less than 0.01 seconds. Note that for all our experiments, we use a Pentium 4 2.8 GHz system with 512 MB RAM running GNU/Linux 2.4.22.

Next, consider the synthetic program consisting of an initial assignment x `:= 0` followed by 1000 increments to x, with the objective of proving that $x = 1000$ at the end. Consider also an alternative version where the program contains only a single loop which increments its counter x 1000 times. We input these two programs to our program verifier, without using abstraction, and to ESC/Java 2 as well. The results are shown in Table 1. For both our verifier and ESC/Java 2 we run both with x initialized to 0 and not initialized, hopefully forcing symbolic execution.

Table 1 shows that our verifier runs faster for the non-looping version. However, there is a noticeable slowdown in the looping version for our implementation. This is caused by the fact that in our implementation of coinductive tabling, subsumption check is done based on similarity of program point. Therefore, when a program point inside a loop is visited for the i-th time, there are $i-1$ subsumption checks to be performed. This results in a total of about 500,000 subsumption checks for the looping program. In comparison, the non-looping version requires only 1,000 subsumption checks. However, our implementation is currently at a prototype stage and our tabling mechanism is not implemented in the most efficient way. For the looping version, ESC/Java 2 employs a weakest precondition propagation calculus; since the program is very small, with a straightforward invariant (just the loop condition), the computation is very fast. Table 1 also shows that there is almost no difference between having x initialized to 0 or not.

Table 1. Timing Comparison with ESC/Java

	Time (in Seconds)			
	CLP with Tabling		ESC/Java 2	
	`x==0`	—	`x==0`	—
Non-Looping	2.45	2.47	9.89	9.68
Looping	22.05	21.95	1.00	1.00

7.2 Experiments Using Abstraction

Next we show an example that demonstrates that the intermittent approach requires fewer predicates. Let us consider a second looping program written in C, shown in Figure 8 (b). The program's postcondition can be proven by providing an invariant

$x=i \wedge i<50$ before the first statement of the loop body of the outer while loop. For predicate abstraction, we use the following predicates x=i, i<50, and respectively their negations $x \neq i$, $i \geq 50$ for that program point to our verifier. The proof process finishes in less than 0.01 seconds. If we do not provide an abstract domain, the verification process finishes in 20.34 seconds. Here intermittent predicate abstraction requires fewer predicates: We also run the same program with BLAST and provide the predicates x=i and i<50 (BLAST would automatically also consider their negations). BLAST finishes in 1.33 seconds, and in addition, it also produces 23 other predicates through refinements. Running it again with all these predicates given, BLAST finishes in 0.28 seconds.

Further, we also tried our proof method on a version of the bakery mutual exclusion algorithm. We need abstraction since the bakery algorithm is an infinite-state program. The pseudocode for process i is shown in Figure 9. Here we would like to verify mutual exclusion, that is, no two processes are in the critical section (program point ⟨2⟩) at the same time. Our version of the bakery algorithm is a concurrent program with asynchronous composition of processes. Nondeterminism due to concurrency can be encoded using nondeterministic choice. We encode the algorithm for 2, 3 and 4 processes in BLAST, where nondeterministic choice is implemented in using the special variable __BLAST_NONDET which has a nondeterministic value. When N is the number of processes, each of the program has the N variables pc_i, where $1 \leq i \leq N$, each denoting the program point of process i. pc_i can only take a value from $\{0,1,2\}$. and also N variables x_i, each denoting the "ticket number" of a process. We also translate the BLAST code into CTS.

```
    while (true) do
⟨0⟩     x_i := max(x_{j≠i}) + 1
⟨1⟩     await (∀j : j ≠ i → x_i<x_j ∨ x_j=0)
⟨2⟩     x_i := 0
    end
```

Fig. 9. Bakery Algorithm Peudocode for

In our experiments, we attempt to verify mutual exclusion property, that is, no two processes can be in the critical section at the same time. Here we perform 3 sets of runs, each consisting of runs with 2, 3 and 4 processes. In all 3 sets, we use a basic set of predicates: $x_i=0$, $x_i \geq 0$, $pc_i=0$, $pc_i=1$, $pc_i=2$, where $i = 1, \ldots, N$ and N the number of processes, and also their negations.

- **Set 1: Use of predicate abstraction at every state with full predicate set**. We perform abstraction at every state encountered during search. In addition to the basic predicates, we also require the predicates shown in Table 2 (a) (and their negations) to avoid spurious counterexamples.
- **Set 2: Intermittent predicate abstraction with full predicate set**. We use intermittent abstraction on our prototype implementation. We abstract only when for some process i, $pc_i=1$ holds. The set of predicates is as in the first set.
- **Set 3: Intermittent predicate abstraction with reduced predicate set**. We use intermittent abstraction on our tabled CLP system. Wee only abstract whenever there are $N-1$ processes at program point 0 (in the 2-process sequential version this means either pc1=0 or pc2=0). For a N-process bakery algorithm, we only need the basic predicates and their negations without the additional predicates shown in Table 2 (a).

Table 2. Results of Experiments Using Abstraction. (a) Additional Predicates. (b) Timing Constraints.

Bakery-2	x1<x2
Bakery-3	x1<x2, x1<x3, x2<x3
Bakery-4	x1<x2, x1<x3, x1<x4 x2<x3, x2<x4, x3<x4

(a)

	Time (in Seconds)			
	CLP with Tabling			BLAST
	Set 1	Set 2	Set 3	
Bakery-2	0.02	0.01	<0.01	0.17
Bakery-3	0.83	0.14	0.09	2.38
Bakery-4	131.11	8.85	5.02	78.47

(b)

We have also compared our results with BLAST. We supplied the same set of predicates that we used in the first and second sets to BLAST. Again, in BLAST we do not have to specify their negations explicitly. Interestingly, for 4-process bakery algortihm BLAST requires even more predicates to avoid refinement, which are `x1=x3+1`, `x2=x3+1`, `x1=x2+1`, 1≤x4, x1≤x3, x2≤x3 and x1≤x2. We suspect this is due to the fact that precision in predicate abstraction-based state-space traversal depends on the power of the underlying theorem prover. We have BLAST generate these additional predicates it needs in a pre-run, and then run BLAST using them. Here since we do not run BLAST with refinement, as the *lazy abstraction* technique [19] has no effect, and BLAST uses all the supplied predicates to represent any abstract state.

For these problems, using our intermittent abstraction with CLP tabling is also markedly faster than both full predicate abstraction with CLP and BLAST. We show our timing results in Table 2 (b) (smallest recorded time of 3 runs each).

The first set and BLAST both run with abstraction at every visited state. The timing difference between them and second and third sets shows that performing abstraction at every visited state is expensive. The third set shows further gain over the second when we understand some intricacies of the system.

Acknowledgement

We thank Ranjit Jhala for help with BLAST.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *15th PLDI*, pages 203–213. ACM Press, May 2001. SIGPLAN Notices 36(5).
2. T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Transactions on Programming Languages and Systems*, 27(2):314–343, 2005.
3. B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. M. Noz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq proof assistant reference manual—version v6.1. Technical Report 0203, INRIA, 1997.
4. A. Bossi, editor. *LOPSTR '99*, volume 1817 of *LNCS*. Springer, 2000.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

6. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
7. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP '92,*, LNCS 631.
8. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In N. Halbwachs and D. Peled, editors, *11th CAV*, number 1633 in LNCS, pages 160–171. Springer, 1999.
9. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 223–239. Springer, 1999.
10. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*. IEEE Computer Society Press, 2000.
11. L. Burdy et. al. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*.
12. Y. S. Ramakrishna et. al. Efficient model checking using tabled resolution. In Grumberg [16].
13. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. R. Ramakrishnan, and U. Ultes-Nitsche, editors, *2nd VCL*, pages 85–96, 2001.
14. L. Fribourg. Constraint logic programming applied to model checking. In Bossi [4], pages 30–41.
15. S. Graf and H. Saïdi. Construction of abstract state graphs of infinite systems with PVS. In Grumberg [16], pages 72–83.
16. O. Grumberg, editor. *CAV '97, Proceedings*, volume 1254 of *LNCS*. Springer, 1997.
17. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *18th RTSS*, pages 230–239. IEEE Computer Society Press, 1997.
18. J. Harrison. HOL light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *1st FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.
19. T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *29th POPL*, pages 58–70. ACM Press, 2002. SIGPLAN Notices 37(1).
20. G. J. Holzmann. *The* SPIN *Model Checker: Primer and Reference Manual*. Add.-Wesley, 2003.
21. J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP($\mathcal{R}$) constraints. In *New Generation Computing*, volume 11, pages 449–469. Ohmsha and Springer-Verlag, 1993.
22. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
23. M. Leuschel and T. Massart. Infinite-state model checking by abstract interpretation and program specialization. In Bossi [4].
24. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *J. Log. and Alg. Prog.*, 58(1–2):89–106, 2004.
25. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
26. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In ed. J. W. Lloyd et. al., editor, *1st CL*, volume 1861 of *LNCS*, pages 384–398. Springer, 2000.
27. S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In D. Kapur, editor, *11th CADE*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

Combining Shape Analyses by Intersecting Abstractions

Gilad Arnold[1], Roman Manevich[2,⋆], Mooly Sagiv[2], and Ran Shaham

[1] University of California, Berkeley
arnold@eecs.berkeley.edu
[2] Tel Aviv University
{rumster, msagiv}@tau.ac.il
ran.shaham@gmail.com

Abstract. We consider the problem of computing the intersection (meet) of heap abstractions.This problem is useful, among other applications, to relate abstract memory states computed by forward analysis with abstract memory states computed by backward analysis. Since dynamically allocated heap objects have no static names, relating objects computed by different analyses cannot be done directly. We show that the problem of computing meet is computationally hard. We describe a constructive formulation of meet based on certain relations between abstract heap objects. The problem of enumerating those relations is reduced to finding constrained matchings in graphs. We implemented the algorithm in the TVLA system and used it to prove temporal heap properties of several small Java programs, and obtained empirical evidence showing the effectiveness of the meet algorithm.

1 Introduction

This research is motivated by the need to approximate temporal properties of programs manipulating dynamically allocated data structures. For example, statically identifying a point in the program after which a list element will never be accessed and thus can be deallocated. As it is undecidable, in general, to prove interesting properties about programs with dynamic memory allocation with pointers and destructive updates, the use of abstract interpretation [2] to compute an over-approximation of a program's operational semantics is a fundamental practice underlying this work. Thus, while proving some correct program properties may fail, every proved property is assured to hold.

We are interested in inferring *persistent* [6] temporal properties of heaps. These are properties that continuously hold from a given point in the trace. Inferring persistent temporal properties is naturally done in two phases, where the first phase over-approximates the shapes of the data structures using forward analysis starting at the entry node, and the second phase computes heap liveness using a backward analysis stating at the exit node. Notice that this generalizes

⋆ This research was supported in part by the Clore Fellowship Programme.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 33–48, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the process of computing scalar liveness in compilers in which the first phase is unnecessary in the absence of pointers and arrays. We call this approach *Phased Bidirectional Analysis*.

The problem of integrating the forward phase with the backward phase is challenging since the exact memory locations are lost by the abstraction. Therefore, this paper addresses the problem of computing the intersection of heap abstractions. When applied to a set of elements of some abstract domain (lattice), this operator—commonly referred to as *meet*—yields the greatest lower bound of the elements in the set. Specifically, for two heap abstractions, the corresponding meet is the set of common stores that are represented by both of its operands.

The main contributions of this paper are summarized as follows:

1. We prove that meet is computationally hard for the abstract domain of bounded structures (Theorem 3), which is used by the TVLA system, by showing a reduction from the problem of 3-colorability on graphs to deciding whether the output of meet is empty. This result is a bit surprising since structures in this domain have unique "canonical names", which makes isomorphism checking and checking of embedding (subsumption) decidable in polynomial time.
2. We present a new algorithm to compute the meet of 3-valued structures. We define the concept of *correspondence* relations between abstract heap objects and explain how to compute meet from these relations. We then develop a strategy to find correspondence relations that manages to prune many of the irrelevant relations thus making the algorithm efficient in practice.
3. We have implemented the meet algorithm in TVLA—a system for generating program analysis from operational semantics [5]—and used it to implement a new analysis for detecting program locations where heap objects and reference fields become unused in Java programs. The information discovered by the analysis can be used to improve memory management. The analysis combines forward and backward information and proves to be precise enough for several small but interesting programs operating on list data structures. The empirical results shows that our analysis is precise enough to reclaim memory as soon as it becomes unneeded. Therefore, our algorithm can serve as a reference algorithm for compile-time garbage collection. Our experiments indicate that the heuristics used by the meet algorithm make it very effective in combining shape analysis; the time and space performance of the algorithm is typically related to the size of the input and output by a linear factor. However, our current prototype implementation is slow and was only applied to small programs.

Running Example. Fig. 1 shows a simple program in a Java-like language that prints the elements of a singly-linked list. This program serves as the running example in this paper. The goal of the analysis here is to discover the earliest points where reference variables and reference fields are no longer used. Specifically, we would like to find that: (a) reference variable `x` is never used after line 7 (this is rather trivial, since `x` does not appear later), and (b) that the reference

```
 [1] x = null;
 [2] while (...) {
 [3]   y = new SLL();
 [4]   y.val = ...;
 [5]   y.n = x;
 [6]   x = y;
     }
 [7] y = x;     // can insert "x = null;" here
 [8] while (y != null) {
 [9]   System.out.print(y.val);
[10]   t = y.n; // can insert "free y;" or "y.n = null;" here
[11]   y = t;
     }
```

Fig. 1. A program that creates a singly-linked list and traverses its elements

field `n` of the object pointed-to by `y` is never used after line 10. The second fact is more challenging to prove, as the object pointed-to by `y` is different on every iteration of the loop.

Outline. The rest of the paper is organized as follows. Section 2 gives an overview of program analysis of heap-manipulating programs using 3-valued logic. In Section 3 we explain how approximate temporal properties of heaps with meet. In Section 4, we present our algorithm for meet. Section 5 describes our experiments with an analyzer that infers compile-time garbage collection information in Java programs by using meet. Section 6 discusses related work.

All proofs, as well as detailed examples, appear in [1].

2 3-Valued Shape Analysis Overview

In this section we explain the representation of concrete program states and their abstractions, based on the parametric analysis framework of [7].

2.1 Concrete Program States

We represent concrete program states by 2-valued logical structures.

Definition 1. *A 2-valued logical structure over a vocabulary (set of predicates) $\mathcal{P}$ is a pair $S = \langle U, \iota \rangle$ where U is the universe of the 2-valued structure, and ι is the interpretation function mapping predicates to their truth-value in the structure: for every predicate $p \in \mathcal{P}$ of arity k, $\iota(p) : U^k \to \{0, 1\}$.*

In this paper, we assume that the set of predicates includes the binary predicate *eq*, and insist that it is interpreted as equality between individuals. Table 1 shows the predicates used to record properties of individuals for the shape analysis of the running example (forward phase).

We denote the set of all 2-valued logical structures over a set of predicates $\mathcal{P}$ by 2-STRUCT[$\mathcal{P}$]. In the sequel, we assume that the vocabulary $\mathcal{P}$ is fixed, and abbreviate 2-STRUCT[$\mathcal{P}$] to 2-STRUCT.

Concrete states (2-valued logical structures) are depicted as directed graphs. Each individual of the universe is drawn as a node. A unary predicate $p(u)$,

Table 1. Predicates used for shape analysis of the running example, and their meaning. The set *PVar* stands for the set of reference variables {x,y,t}.

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	Is v_1 equal to v_2?
$\{x(v) : x \in PVar\}$	Does reference variable `x` point to object v?
$n(v_1, v_2)$	Does the `n` field of object v_1 point to object v_2?
$\{r_{x,n}(v) : x \in PVar\}$	Is v reachable from reference variable `x` along `n` fields?
$is(v)$	Do two or more fields of heap elements point to v?
$c_n(v)$	Is v on a directed cycle of `n` fields?

which holds for an individual u, appears next to the corresponding node. If a unary predicate represents a reference variable, it is shown by having an arrow drawn from its name to the node referenced by the variable. The binary predicate $n(u_1, u_2)$, which holds for a pair of individuals u_1 and u_2, is drawn as a directed edge from u_1 to u_2, and labeled n. The predicate eq is not drawn, since any two nodes are different and every node is equal to itself.

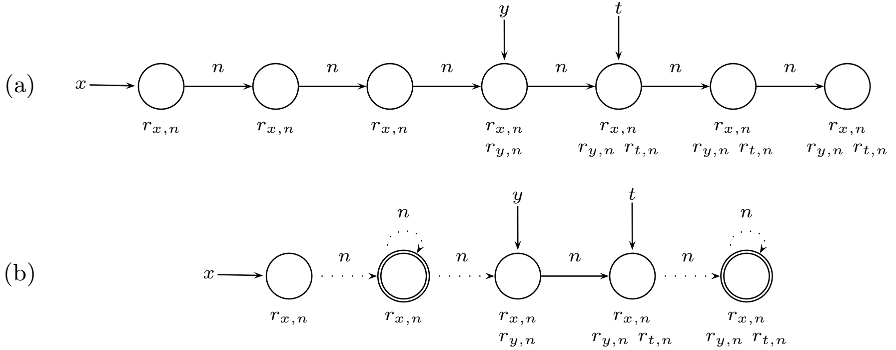


Fig. 2. (a) A concrete program state arising after the execution of the statement $t = y.n$; (b) An abstract program state approximating the concrete state in (a)

Fig. 2(a) shows a concrete program state arising after the execution of the statement $t = y.n$ on line 10 of the running example in Fig. 1.

2.2 Abstract Program States

The abstract program states we use are based on 3-valued logic [7], which extends boolean logic by introducing a third value 1/2, denoting values that may be either 0 or 1. In particular, we utilize the partially ordered set $\{0, 1, 1/2\}$ where $0 \sqsubseteq 1/2$ and $1 \sqsubseteq 1/2$, with the join operation $\sqcup$, defined by $x \sqcup y = x$ if $x = y$, and $x \sqcup y = 1/2$ otherwise.

Definition 2. *A* 3*-valued logical structure over a set of predicates* $\mathcal{P}$ *is a pair* $S = (U, \iota)$ *where* U *is the universe of the* 3*-valued structure, and* ι *is the interpretation function mapping predicates to their truth-value in the structure: for every predicate* $p \in \mathcal{P}$ *of arity* k, $\iota(p) : U^k \to \{0, 1, 1/2\}$.

An abstract state may include summary nodes*, i.e., an individual which corresponds to one or more individuals in a concrete state represented by that abstract state. A summary node u has $eq(u, u) = 1/2$, indicating that it may represent more than a single individual.*

Abstract states (3-valued logical structures) are also depicted as directed graphs, where unary predicates denoting reference variables, as well as binary predicates, with 1/2 values are shown as dotted edges. Summary individuals appear as double-circled nodes. A unary predicate that evaluates to 1/2 for a node is depicted by having = 1/2 next to the name of the predicate.

We denote the set of all 3-valued logical structures over a set of predicates $\mathcal{P}$ by 3-STRUCT[$\mathcal{P}$], and usually abbreviate it to 3-STRUCT.

We define a partial order on structures, denoted by $\sqsubseteq$, based on the concept of *embedding.*

Definition 3 (Embedding). *Let $S = (U, \iota)$ and $S' = (U', \iota')$ be two structures and let $f : U \to U'$ be a surjective function. We say that f* embeds *S in S', denoted $S \sqsubseteq^f S'$, if for every predicate $p \in \mathcal{P}^{(k)}$ and k individuals $u_1, \ldots, u_k \in U$,*

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \ . \tag{1}$$

We say that S is embedded *in S', denoted $S \sqsubseteq S'$, if there exists a function f such that $S \sqsubseteq^f S'$. We also say that S'* approximates *S.*

The embedding order is used to define a concretization function for a single 3-valued structure S by $\sigma(S) = \{S' \in \text{2-STRUCT} \mid S' \sqsubseteq S\}$. The concretization of a set of 3-valued structures is defined by $\gamma(XS) = \bigcup_{S \in XS} \sigma(S)$.

The embedding order induces a Hoare preorder on sets of 3-valued structures.

Definition 4. *For sets of structures $XS_1, XS_2 \subseteq$ 3-STRUCT, $XS_1 \sqsubseteq XS_2$ if and only if $\forall S_1 \in XS_1 : \exists S_2 \in XS_2 : S_1 \sqsubseteq S_2$.*

In the following definition, we restrict sets of 3-valued structures by disallowing non-maximal structures. This ensures that the Hoare ordering is a proper partial ordering on the sets.

We are now ready to present the abstract domain which is considered for the construction of the meet algorithm.

Definition 5 (Core Abstract Domain). *The abstract domain $D_{\text{3-STRUCT}}$ consists of all sets of 3-valued structures that do not contain non-maximal structures, $\{XS \subset \text{3-STRUCT} \mid \forall S_1, S_2 \in XS : S_1 \sqsubseteq S_2 \implies S_1 = S_2\}$, with the same ordering as in Definition 4.*

2.3 Bounded Program States

Note that the size of a 3-valued structure is potentially unbounded and that 3-STRUCT is infinite. The abstractions studied in [7], and also used for the analysis in Section 5, rely on a fundamental abstraction function for converting

a potentially unbounded structure—either 2-valued or 3-valued—into a bounded 3-valued structure.

A 3-valued structure is said to be *bounded* if for every two distinct individuals in its universe there exists a unary predicate p such that either $p^{S_1}(u_1) = 0$ and $p^{S_2}(u_2) = 1$ or $p^{S_1}(u_1) = 1$ and $p^{S_2}(u_2) = 0$.[1] We denote the set of all bounded 3-valued structures over a set of predicates $\mathcal{P}$ by B-STRUCT[$\mathcal{P}$]. The finite abstract domain $D_{\text{B-STRUCT}}$ is a sublattice of $D_{\text{3-STRUCT}}$, containing all sets of bounded structures that do not contain non-maximal structures.

The abstraction function $\beta^{\mathcal{P}}_{blur}$: 2-STRUCT[$\mathcal{P}$] $\rightarrow$ B-STRUCT[$\mathcal{P}$] converts a (potentially unbounded) 2-valued structure into a bounded 3-valued structure, by merging all individuals with the same values for all unary predicates. Namely, $\beta^{\mathcal{P}}_{blur}((U, \iota)) = (U', \iota')$, where U' is the set of equivalence classes in U of nodes with same values for all unary predicates, and the interpretation ι' of each predicate $p \in \mathcal{P}^{(k)}$ and k individuals $c_1, \ldots, c_k \in U'$ is given by

$$p^{S}\ (c_1, \ldots, c_k) = \bigsqcup_{u_i \in c_i} p^{S}(u_1, \ldots, u_k)\ .$$

Fig. 2(b) shows a bounded structure obtained from the structure in Fig. 2(a).

The abstraction function β_{blur}, which is called *canonical abstraction*, serves as the basis for abstract interpretation in TVLA [5]. In particular, it serves as the basis for defining various different abstractions for the (potentially unbounded) set of 2-valued logical structures that may arise at a program point, by defining different sets of predicates. We also define the function α, which extends β_{blur} to sets of structures: $\alpha(XS) = \bigsqcup\{\beta_{blur}(S) \mid S \in XS\}$.[2]

3 Inferring Temporal Properties Via Staged Bidirectional Analysis

Persistent temporal properties can be efficiently verified without explicitly representing traces. An example of such a property is liveness of reference variables and reference fields. A reference variable or reference field is said to be *dead* (i.e., not *live*) at a given program point if on every execution that goes through that point it is not used before being redefined.

The (possibly infinite) set of temporal properties is defined as the least fixed point of the following (not necessarily computable) system of equations:

$$\begin{aligned}
\overrightarrow{CS}_{entry} &= CS_{init}\\
\overrightarrow{CS}_{l_2} &= \left\{S_{out} \,\middle|\, (l_1, l_2) \in E,\ S_{in} \in \overrightarrow{CS}_{l_1},\ (l_1, S_{in}) \rightsquigarrow (l_2, S_{out})\right\}\\
\overleftarrow{CS}_{exit} &= CS_{final} \cap \overrightarrow{CS}_{exit}\\
\overleftarrow{CS}_{l_1} &= \left\{S_{in} \,\middle|\, (l_1, l_2) \in E,\ S_{out} \in \overleftarrow{CS}_{l_2},\ (l_1, S_{out}) \leftrightsquigarrow (l_2, S_{in})\right\} \cap \overrightarrow{CS}_{l_1}\ .
\end{aligned}$$

[1] The notion of a bounded structure can be generalized by considering any subset of the set of unary predicates, as done in TVLA.

[2] The operator $\bigsqcup$ is the least upper bound operator in $D_{\text{B-STRUCT}}$.

Here, it is assumed that the concrete 2-valued structures also record information on temporal properties that hold on program executions. The program is represented as a control flow graph, with entry and exit nodes *entry* and *exit*, respectively, and a set of control flow edges E. CS_{init} is the initial set of concrete stores at the entry location including all possible values associated with temporal properties. CS_{final} represents the set of states in which all temporal properties are set to their final values (that is, their values upon termination of the execution). We write $(l_1, S_{in}) \rightsquigarrow (l_2, S_{out})$ to denote the transformation induced by the forward execution of the statement or condition at edge (l_1, l_2). Program conditions are interpreted according to the standard semantics. Note that the forward semantics sets values non-deterministically to the temporal properties predicates. We write $(l_1, S_{out}) \leftrightsquigarrow (l_2, S_{in})$ to denote the transformation induced by the backward execution of the statement or condition at edge (l_1, l_2). This semantics sets the values of the changed temporal properties. Variables whose values are changed are updated non-deterministically.

The above system of equations does not necessarily terminate for programs with loops. Therefore, an upper approximation to this system is conservatively computed by representing sets of states using 3-valued structures. Extra predicates store values of tracked temporal properties. Moreover, the ability to define unary predicates allows tracking of an unbounded number of temporal properties. Both forward and backward executions are conservatively executed on 3-valued structures. However, as backward reasoning uses results obtained by the forward counterpart, it is considered a *secondary stage* taking place after the forward reasoning is complete. Finally, intersection ($\cap$) is over-approximated using meet ($\sqcap$).

3.1 Compile-Time GC Analysis

We now explain how compile-time garbage collection information can be computed using a phased bidirectional verification.

In particular, we are interested in identifying the first point in the trace where an object is not further used, and therefore may be safely deallocated by a `free` statement. Thus, the backward execution of a statement tracks the use of objects. Our analysis maintains the predicate $use(v)$ to track object future usage information.

An object v is denoted *used* in a statement or a condition at edge (l_1, l_2), if a reference expression e, that evaluates to v, is used for dereference at that statement. In such a case, the backward execution of the statement $(l_1, S_{out}) \leftrightsquigarrow (l_2, S_{in})$ records in S_{in} the fact that v is used by setting $use(v)$ to 1. As mentioned, the forward execution of a statement non-deterministically sets values to $use(v)$.

Fig. 3(a) shows one of the structures that arise before the statement `t = y.n` at line 10 of Fig. 1, and Fig. 3(b) shows one of the structures that arise after that statement. The object referenced by `y` is still used before the statement, as $use(v)$ holds for the individual referenced by `y`. Nonetheless, the object referenced by `y` is not (further) used after that statement, as $use(v)$ does not hold for the individual referenced by `y`. Having verified that $use(v)$ does not hold for any

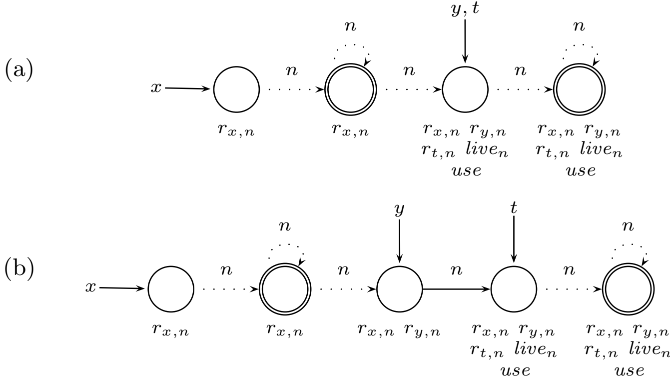


Fig. 3. 3-valued structures representing sets of program configurations, including heap object and reference field liveness, that arise (a) before the execution of the statement `t = y.n`; and (b) after it is executed

individual v referenced by `y`, for all structures that may arise after the aforementioned statement, we conclude that `free y` may be inserted after the statement `t = y.n`, to deallocate the object referenced by `y`, as it is no longer used in the program. Moreover, since *for all structures* arising before that statement, the object referenced by `y` is still used, placing a `free y` after that statement will free the space referenced by `y` at the earliest possible time.

3.2 Assign-Null Analysis

Another application of phased bidirectional analysis is the computation of heap reference liveness, providing for compile-time optimization of runtime garbage collection effectiveness. For each object reference field, we identify whether it is *live* at any point in the trace, meaning that it may be used, prior to being redefined, after that point. We are interested in spotting points in the trace where a reference field becomes *dead*, and therefore may be assigned a `null` value, thus significantly reducing potential GC drag time [8]. Here, again, the backward execution of the statement tracks the uses (dereference) and redefinitions (assignment) of object fields. In particular, for each reference field f which is a member of some object v, the predicate $live_f(v)$ is used to record future use and re-definition information (in our example f is n).

A reference field f of an object v is denoted *used* in a statement or a condition at edge (l_1, l_2) if an expression e—which is not an l-value—refers to the value of f. In this case, the backward execution of the statement $(l_1, S_{out}) \leftsquigarrow (l_2, S_{in})$ sets $live_f(v)$ to 1. Otherwise, f is denoted *redefined* if it is being assigned a new value, namely, being referred to by an l-value expression e. In this case, the backward execution of the statement sets $live_f(v)$ to 0. Here as well, forward execution non-deterministically sets values to $live_f(v)$.

Section 5 includes experimental results for an implementation of both of the analyses described in this section.

4 Computing the Meet of Heap Abstractions

In this section, we develop a meet algorithm for a family of abstract domains and discuss the complexity of the algorithm for two cases: (i) for arbitrary 3-valued structures, and (ii) for bounded structures.

4.1 The Problem Setting

Our aim is to provide an algorithm applicable for a family of abstract domains based on 3-valued structures, including the abstract domain of bounded structures, $D_{\text{B-STRUCT}}$.

We design a meet algorithm for the domain $D_{\text{3-STRUCT}}$, which we consider as a basis for other abstract (sub-) domains. Given a sub-domain $\mathcal{D} \subseteq D_{\text{3-STRUCT}}$ and a set of abstract elements $X \in \mathcal{D}$, the result of the algorithm is possibly not an element of $\mathcal{D}$. However, when $\bigsqcap_{\mathcal{D}} X$ is defined, the inequality $\bigsqcap_{\mathcal{D}} X \sqsubseteq \bigsqcap_{D_{\text{3-STRUCT}}} X$ holds. Therefore, a domain specific operator $\mathit{Refine}_{\mathcal{D}} : D_{\text{3-STRUCT}} \to \mathcal{D}$ can be used to refine the result to yield an element of $\mathcal{D}$, $\mathit{Refine}_{\mathcal{D}}(\bigsqcap_{D_{\text{3-STRUCT}}} X) = \bigsqcap_{\mathcal{D}} X$.

For certain abstract domains, including $D_{\text{B-STRUCT}}$, no refinement is required. We now explain this formally.

Definition 6. *We say that an abstract domain $\mathcal{D} \subseteq D_{\textit{3-STRUCT}}$, with the same ordering between abstract elements as in $D_{\textit{3-STRUCT}}$ (see Definition 4), is* meet-admissible *when it satisfies the following conditions.*

Sublattice of $D_{\text{3-STRUCT}}$. *$\mathcal{D}$ is a lattice, and $\bigsqcap_{\mathcal{D}} X = \bigsqcap_{D_{\textit{3-STRUCT}}} X$ and $\bigsqcup_{\mathcal{D}} X = \bigsqcup_{D_{\textit{3-STRUCT}}} X$ for every finite subset X of $\mathcal{D}$.*

Closure of singletons. *For every structure $S \in$ 3-STRUCT, if S exists in some set $XS \in \mathcal{D}$ then $\{S\} \in \mathcal{D}$. This condition allows us to break the problem of computing meet on sets of structures to a set of sub-problems where meet is computed on pairs of structures.*

Theorem 1. *The (parametric) abstract domain of bounded structures, $D_{\textit{B-STRUCT}}$, is meet-admissible.*

The following proposition reduces the problem of computing the meet of two sets of structures to the problem of computing the meet of two structures by using the join operator, which we discuss at the end of this section.

Proposition 1. *Let XS_1, XS_2 be two elements in a meet-admissible domain $\mathcal{D}$. Then,*

$$XS_1 \sqcap XS_2 = \bigsqcup_{\substack{S_1 \in XS_1 \\ S_2 \in XS_2}} \{S_1\} \sqcap \{S_2\} \ . \tag{2}$$

In the remainder of this section, we consider the following problem. Given two structures $S_1, S_2 \in$ 3-STRUCT, compute $\{S_1\} \sqcap \{S_2\}$.

4.2 Computing the Meet of Two Structures

Fig. 4 shows two structures and their meet. (For now, ignore the edges between the structures in Fig. 4(a) and Fig. 4(b).) The structure in Fig. 4(a) arises during forward shape analysis, after the statement $t = y.n$ at line 10 of the running example; this is the structure from Fig. 2(b) with non-deterministic assignments to the values of the predicates $puse(v)$ and $live_n(v)$. The structure in Fig. 4(b) is obtained from the structure in Fig. 3(b) by backward execution of the statement $y = t$ at line 11 of the running example. The meet of these two structures results in the structure shown in Fig. 4(c).

We now establish a connection between the structures that comprise the result of meet and certain relations that hold between their individuals. We first define the meet of two Kleene values t_1 and t_2. If $t_1 \sqsubseteq t_2$ then $t_1 \sqcap t_2 = t_1$, if $t_2 \sqsubseteq t_1$ then $t_1 \sqcap t_2 = t_2$, and otherwise the result is undefined and we denote it by the special symbol $\perp$.

Definition 7 (Meet Correspondence). *Given two structures $S_1 = (U_1, \iota_1)$ and $S_2 = (U_2, \iota_2)$, a relation $M \subseteq U_1 \times U_2$ is a* meet correspondence *between S_1 and S_2 when it is: (a)* Full*, i.e.,*

$$\forall u_1 \in U_1 : \exists v_2 \in U_2 : u_1\, M\, v_2 \;\; and \; \forall v_2 \in U_2 : \exists u_1 \in U_1 : u_1\, M\, v_2 \;\; ;$$

and (b) Consistent*, i.e., for every predicate p of arity k, and a pair of k-tuples $u_1, \ldots, u_k \in {U_1}^k$ and $v_1, \ldots, v_k \in {U_2}^k$, such that $u_i\, M\, v_i$ for $i = 1 \ldots k$, $p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \neq \perp$.*

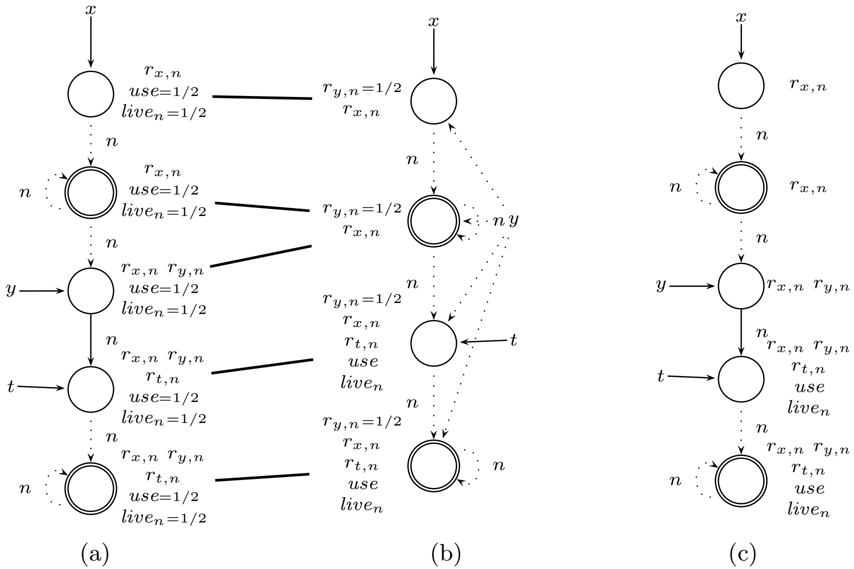


Fig. 4. An example for computing meet for the running example. (a) A structure that arises during the forward shape analysis; (b) A structure that arises during the backward (object liveness) analysis. (c) The meet of (a) and (b).

The structures in Fig. 4(a) and Fig. 4(b) have exactly one meet correspondence, which is shown by the edges between their individuals.

We can use a meet correspondence to construct a common lower bound of two structures in the following way.

Definition 8. *Given a meet correspondence M between structures $S_1 = (U_1, \iota_1)$ and $S_2 = (U_2, \iota_2)$, the operation $S_1 \sqcap_M S_2$ yields the M-*induced *structure $S = (U, \iota)$, where $U = \{\langle u, v\rangle \in M\}$, and the interpretation of every predicate p of arity k and every k-tuple of nodes $\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle \in U^k$ is given by*

$$p^S(\langle u_1, v_1\rangle, \ldots, \langle u_k, v_k\rangle) = p^{S_1}(u_1, \ldots, u_k) \sqcap p^{S_2}(v_1, \ldots, v_k) \ .$$

We are now ready to characterize the result of the meet operator in terms of meet correspondences.

Theorem 2. *Let $\mathcal{M}_{S_1,S_2} \subseteq \wp(U_1 \times U_2)$ denote the set of meet correspondences between structures S_1 and S_2. Then,*

$$\{S_1\} \sqcap \{S_2\} = \bigsqcup_{M \in \mathcal{M}_{S_1,S_2}} \{S_1 \sqcap_M S_2\} \ .$$

Theorem 2 already gives us a naive way to compute meet by: (a) Enumerating all relations $M \in U_1 \times U_2$; (b) Checking each of them to see whether it constitutes a meet correspondence; (c) For each meet correspondence, computing $S_1 \sqcap_M S_2$, and (d) Combining the results via join. Although the meet of two structures is a set of structures containing $2^{|U_1 \times U_2|}$ structures in the worst case, the size of the set is usually small, in practice. Notice that the above approach is intractable even when the number of structures is small, since the majority of the relations are not meet correspondences.

An immediate consequence of [10] is that deciding whether the meet of two arbitrary 3-valued structures is empty is NP-complete. The next theorem states that meet is computationally hard even for two bounded structures.

Theorem 3. *Given two bounded structures S_1 and S_2, the problem of deciding whether $\{S_1\} \sqcap \{S_2\} \neq \emptyset$ is NP-complete.*

Since the problem of computing meet with polynomial worst-case complexity is hard, we aim to achieve good efficiency in practice. We develop an algorithm based on a strategy. The strategy exploits certain properties of the abstract domain to prune the set of relations and find the meet correspondences. In Section 5, we supply empirical evidence showing that the algorithm successfully prunes most irrelevant relations when used in an abstract interpreter for inferring temporal heap properties on several benchmark programs.

4.3 Enumerating Meet Correspondences

We now present a strategy for exploring the (exponential) space of relations between two structures, searching for meet correspondences. The strategy, shown

in pseudo-code in [1], attempts to prune relations that do not constitute a meet correspondence as much as possible, and relies on another procedure for solving a graph-matching problem on graphs (explained below).

The strategy consists of 4 stages that are run consecutively:

1. **Consistency of nullary predicates.** If there exists a nullary predicate p such that $p^{S_1}() = 1$ and $p^{S_2}() = 0$ or $p^{S_1}() = 0$ and $p^{S_2}() = 1$, then the result of meet is the empty set.
2. **Removing infeasible node pairs.** We remove from the set $U_1 \times U_2$ node pairs $\langle u, v\rangle$ such that there exists a predicate p of arity k and $p^{S_1}(u^k) \sqcap p^{S_2}(v^k) = \bot$, where u^k denotes a k-tuple containing the node u in all k positions. By Definition 7 these pairs are not contained in any meet correspondence.
3. **Finding full relations.** To satisfy the fullness requirement of Definition 7, we solve the following graph matching problem. Given a graph $G = \langle V, E\rangle$ and a subset W of V, find all subsets $M \subseteq E$ such that in the graph $\langle V, M\rangle$ the degree of every vertex is at least 1, and for vertices in W the degree is at most 1. In our case, $V = U_1 \cup U_2$, E is the set of pairs from the previous stage, and W is the set of non-summary nodes. An (worst-case exponential time) algorithm for this problem that uses several heuristics to solve this problem efficiently is discussed in [1].
4. **Consistency test.** The full relations from the previous stage are tested for consistency according to Definition 7 (in polynomial time). The relations that pass the test are meet correspondences and are used to create M-induced structures which are combined via join to yield the result.

The intuition behind the second stage is that two structures, possible produced by different analyses, may share a common set of unary predicates that are assigned only definite values, i.e., 0 or 1. Usually, these are the predicates that represent reference variables. In such cases, these predicates help prune many of the infeasible edges and determine a subset of edges with degree 1, which must participate in every meet correspondence. Our algorithm uses these edges to reduce the amount of searching that has to be done.

In Fig. 4, the first stage of the algorithm is degenerate, as there are no nullary predicates. The second stage prunes the set of all node pairs, which consists of 20 pairs, to 5. This reduction occurs since the predicates x, t, $r_{x,n}$, and $r_{t,n}$ have definite values in both structures. In this example, there is only one full relation, which is returned by the third stage of the algorithm. This relation is indeed consistent, and thus the structure in the output is produced.

4.4 Computing Join

The join of sets of 3-valued structures is set union, followed by removal of non-maximal structures. To remove non-maximal structures, we need an algorithm to check for whether a structure $S_1 = (U_1, \iota_1)$ is embedded in a structure $S_2 = (U_2, \iota_2)$.

We observe that an embedding relation (see Definition 3) is actually a meet correspondence that satisfies a stricter version of the consistency condition. It

is: (i) Full; and (ii) for every predicate p of arity k, and a pair of k-tuples $u_1, \ldots, u_k \in {U_1}^k$ and $v_1, \ldots, v_k \in {U_2}^k$, such that $u_i \, M \, v_i$ for $i = 1 \ldots k$, $p^{S_1}(u_1, \ldots, u_k) \sqsubseteq p^{S_2}(v_1, \ldots, v_k)$.

Since checking the second condition for two structures can be done in polynomial time, we can reuse the techniques for finding meet correspondences. In the first stage we check that for every nullary predicate p, $p^{S_1}() \sqsubseteq p^{S_2}()$. In the second state we remove from the set $U_1 \times U_2$ all node pairs $\langle u, v \rangle$ such that there exists a predicate p of arity k and $p^{S_1}(u^k) \not\sqsubseteq p^{S_2}(v^k)$. We then proceed by enumerating full relations over the remaining node pairs to find one that fulfills the second condition of the embedding relation.

For arbitrary 3-valued structures, checking embedding is NP-complete. However, for bounded structures our algorithm decides the problem in polynomial time. This is because, for two bounded structures, an embedding relation, if one exists, is unique and completely determined by the unary predicates.

5 Inferring Temporal Properties for Compile-Time Memory Management

Compile-time GC is most desirable for lightweight Java-based platforms, such as JavaCard, where the penalty induced by a runtime GC is sometimes intolerable due to the limited space and processing power. Such platforms normally provide a mechanism for explicit memory deallocation, e.g., through a free directive.

We have implemented the phased bidirectional analysis described in Section 3 in the TVLA system to infer compile-time GC information. Our analysis infers information for producing a set of free statements that can be safely added to the program to free unused objects. Moreover, our analysis ensures that an object is deallocated at the earliest possible time, i.e., immediately after the object is last used.

5.1 Experimental Results

Table 2 shows our benchmark programs, which were used in [9].[3] The first four programs involve manipulations of singly-linked lists. `DLoop` and `DPairs` involve manipulations of doubly-linked lists. The `small-javac` example was used in [8], where it has been shown that a significant potential for compile-time GC exists by manually rewriting the code to include null assignments. Our assign-null analysis is able to yield the manual rewriting automatically.

On all benchmark programs, both our compile-time GC and assign-null analyses were able to detect all opportunities for object deallocation and safe assignment of null to reference fields, respectively. This information allows the reclamation of unused space at the earliest possible time. For example, considering the program in Fig. 1, the compile-time GC analysis was able to determine the safe deallocation of the object pointed by y right after line 10, thus deallocating list elements as soon as they are being traversed. Our assign-null analysis was

[3] The programs are available from www.cs.tau.ac.il/~rumster/ctgc_benchmarks.zip.

Table 2. Benchmarks and analysis costs (in seconds and Mb.)

Program	Description	Forward		Backward	
		Time	Space	Time	Space
`Loop`	Running example (Fig. 1)	0.9	1.0	1.6	1.8
`CReverse`	Constructive list reversal	3.0	2.0	5.7	4.2
`Delete`	Deletion of a list element	12.4	3.2	41.1	12.9
`DLoop`	Doubly-linked list variant of `Loop`	1.4	1.3	2.3	2.5
`DPairs`	Doubly-linked list traversal in pairs	3.0	2.0	5.5	4.0
`small-javac`	Emulation of JavaC's parser facility	528.9	32.1	334.4	77.6

able to verify that a `y.n=null` assignment could be inserted after line 10. The analyses proved similar properties for the other benchmark programs.

Table 2 shows the costs of the analysis on the benchmark programs. As both analyses have very similar costs, we only show the results of the compile-time GC analysis.

The experiments were conducted on a 1.6 GHz laptop with 512 Mb. of memory, running Windows XP.

In addition to analysis time and space, we measured two redundancy factors related to our meet algorithm. We evaluated the efficiency of the graph matching algorithm in stage 3. The results show that for all benchmark programs, at most 0.5% of the expanded search space did not lead to valid matchings. We also measured the percentage of full relations computed during stage 3 of the algorithm that did not constitute meet correspondences (eliminated in stage 4). In all benchmarks the average number of relations that were eliminated did not exceed 0.3%, and in most benchmarks no eliminations occurred.

We believe that our meet algorithm is efficient for the the following reason. The forward shape analysis produces very precise information, which means that the values of the unary shape predicates (in Table 1) are almost always definite. When the backward phase computes the backward effect of a statement, it accepts a structure where all unary predicates are definite and assigns non-deterministic values to only a fixed number of unary predicates—y and $r_{n,y}$ in the structure shown in Fig. 4(b). Then, the meet is applied to structures where most unary shape predicates predicates have definite values. Our algorithm is geared to exploit these situations by focusing the search for meet correspondences.

6 Related Work

Computing Meet of Heap Abstractions. In [3], a meet is used for interprocedural shape analysis. Two algorithms are presented for computing meet on bounded structures. The first algorithm uses a "canonicalization"[4] operation to transform the structures to sets of structures in the image of canonical abstraction with the same concretization. Computing meet for the resulting structures is then straightforward. However, canonicalization can unnecessarily increase the number of structures by an exponential factor in the worst case. In our examples the worst case would indeed manifest itself, since we set the values of the

[4] Canonicalization is a semantic reduction akin to substituting abstract elements by their respective set of join-irreducibles.

temporal heap properties to non-deterministic values. Our algorithm avoids this problem by operating directly on the given structures. The second algorithm approximates meet by transforming one of the structures into a dynamic set of constraints and using a constraint solver. While usually more efficient than the first algorithm, it computes an over-approximation of meet. We believe that our algorithm can be used to improve the running times of the interprocedural analysis reported in [3].

In [4], it is shown how to compute meet for a class of formulas that precisely characterize bounded structures. The computation is essentially achieved in the same way as the first algorithm in [3].

In [11], a symbolic semi-algorithm for meet is presented. The algorithm converts bounded structures to formulas, and then uses logical conjunction to compute the result in the domain of formulas. Converting the resulting formula back to bounded structures is done via a theorem prover. The algorithm operates with respect to a finer concretization function than the one defined in Section 2. Specifically, this concretization function is parameterized by a set of integrity constraints C, and is defined by

$$\gamma_C(S) = \left\{ S' \in \text{2-STRUCT} \mid S' \sqsubseteq S,\, S' \models C \right\} .$$

The advantage of this algorithm is that it provides the most precise result with respect to γ_C. However, its performance can be quite low, due to the use of canonicalization and a potentially large number of calls to a theorem prover.

A distinct advantage of the algorithm presented in this paper is that it is not restricted to bounded structures and works for any set of 3-valued structures.

Compile-Time Memory Management. Most of the work on compile-time GC analysis has been done for functional languages. This paper demonstrates a compile-time GC analysis that applies to an imperative language with destructive updates, and is capable of reclaiming an object that is still reachable, but not used further in the run.

In [9], a user-specification-driven compile-time GC and assign-null analysis are described. The user specifies a *free query* of the form (pt, x), where pt is a program location and x is a program variable. A positive answer to the query means that a `free x` statement may be issued after program point pt. In contrast, the algorithms in this paper do not require nor rely on a user-specified queries, but rather perform an analysis on an exhaustive set of queries generated automatically using a simple heuristic. We believe our approach may be significantly more efficient compared to the analysis of [9] with our exhaustive set of queries.

References

1. G. Arnold, R. Manevich, M. Sagiv, and R. Shaham. Intersecting heap abstractions with applications to compile-time memory management. Technical Report TR-2005-04-135520, Tel Aviv University, Apr 2005. Available at http://www.cs.tau.ac.il/~rumster/TR-2005-04-135520.pdf.

2. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.
3. B. Jeannet, Alexey L., T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proc. Static Analysis Symp.* Springer, 2004.
4. V. Kuncak and M. Rinard. Boolean algebra of shape analysis constraints. In *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, pages 59–72. Springer, January 2004.
5. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symp.*, pages 280–301, 2000.
6. Z. Manna and A. Pnueli. A hierarchy of temporal properties (invited paper). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410, 1989.
7. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
8. R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM Press, June 2001.
9. R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proc. of Static Analysis Symposium (SAS'03)*, volume 2694 of *LNCS*, pages 483–503. Springer, June 2003.
10. G. Yorsh. Logical characterizations of heap abstractions. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2003. http://www.cs.tau.ac.il/~gretay/.
11. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference (TACAS 2004)*, pages 530–545. Springer, March 2004.

A Complete Abstract Interpretation Framework for Coverability Properties of WSTS*

Pierre Ganty[1,**], Jean-François Raskin[1], and Laurent Van Begin[2]

[1] Département d'Informatique, Université Libre de Bruxelles
{pganty, jraskin}@ulb.ac.be
[2] LIAFA, Université Paris 7
lvbegin@liafa.jussieu.fr

Abstract. We present an abstract interpretation based approach to solve the coverability problem of well-structured transition systems. Our approach distinguishes from other attempts in that (1) we solve this problem for the whole class of well-structured transition systems using a forward algorithm. So, our algorithm has to deal with possibly infinite downward closed sets. (2) Whereas other approaches have a non generic representation for downward closed sets of states, which turns out to be hard to devise in practice, we introduce a generic representation requiring no additional effort of implementation.

1 Introduction

Model-checking is nowadays widely accepted as a powerful technique for the automatic verification of reactive systems that have natural finite state abstractions. However, many reactive systems are only naturally modeled as infinite-state systems. This is why a large research effort was done in the recent years to allow the direct application of model-checking techniques to infinite-state models. This research line has shown successes for several interesting classes of infinite-state systems, for example: timed automata [1], hybrid automata [2], FIFO channel systems [3, 4], extended Petri nets [5, 6], broadcast protocols [7], etc.

General decidability results hold for a large class of infinite-state systems called the *well-structured transition systems*, WSTS for short. WSTS are transition systems whose sets of states are well-quasi ordered and whose transition relations enjoy a monotonicity property with respect to the well-quasi order. Examples of WSTS are Petri nets [8], monotonic extensions of Petri nets (Petri nets with transfer arcs [9], Petri nets with reset arcs [10], and Petri nets with non-blocking arcs [11]), broadcast protocols [12], lossy channel systems [3]. For all those classes of infinite-state systems, we know that an interesting and large class of *safety properties* are decidable by reduction to the *coverability problem*. The coverability problem is defined as follows: "given a WSTS for the well-quasi

* Supported by the FRFC project "Centre Fédéré en Vérification" funded by the Belgian National Science Foundation (FNRS) under grant nr 2.4530.02.
** Pierre Ganty is supported by the FNRS under a FRIA grant.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 49–64, 2006.
© Springer-Verlag Berlin Heidelberg 2006

order $\succeq$, and two states c_1 and c_2, does there exist a state c_3 which is reachable from c_1 and such that $c_3 \succeq c_2$?" (in that context, we say that c_3 covers c_2).

Broadly speaking, there are two ways to solve the coverability problem for WSTS. The first way to solve the coverability problem is to explore *backwardly* the transition system by iterating the *pre* operator[1] starting from the set of states that are greater or equal to c_2. This simple procedure is effective when very mild assumptions are met. In fact, for any well-quasi ordered set $(X, \succeq)$, the following nice property holds: every $\succeq$-upward closed[2] set can be *finitely* represented using its finite set of minimal elements[3]. This generic representation of $\succeq$-upward closed set is adequate as union and inclusion are effective. The only further property that is needed for the procedure to be effective is that given a finite set of minimal elements M defining an $\succeq$-upward closed set U, it must be possible to compute the finite set of minimal elements M' representing $pre(U)$. Higman's lemma [13] on well-quasi orders ensure the termination of this procedure.

The second way is to explore forwardly the transition system from the initial state c_1. Here, the situation is more complicated. A saturation method that iterates the *post* operator[4] from c_0 can not lead to an algorithm as the reachability problem is undecidable for WSTS. Recently, we have shown that the coverability problem can be decided in a forward way by constructing two sequences of abstractions of the reachable states of the system, one from below and one from above [14]. The sequence of abstractions from below allows us to detect positive instances of the coverability problem and it is simply the bounded iteration of *post* from the initial state. The abstraction from above is the iteration of an overapproximation of *post* over downward closed set of states that becomes more and more precise. This sequence allows us to decide negative instances of the problem. This schema of algorithm is general but to be applicable to a given class of WSTS, the user has to provide a, so called, *adequate domain of limits*. This set is in fact a (usually infinite) set of abstract values that allows to represent any downward closed set. The situation is less satisfactory than for upward closed set where there exists, as we have seen above, a simple and generic way to represent upward closed set by sets of minimal elements. Such a generic way of representing downward closed sets was missing and this problem is solved here.

The contributions of this paper are as follows. First, we show that for any well-quasi ordered set, there exists a generic and effective representation of downward closed sets. To the best of our knowledge, this is the first time that such a generic representation is proposed. An attempt in that direction was taken in [15] but the result is a theory for designing symbolic representation of downward closed sets

[1] A function that returns all the states that have a one-step successor in a given set of states.

[2] A set S is upward (resp. downward) closed if for any c such that $c \succeq s$ (resp. $c \preceq s$) for some $s \in S$ we have $c \in S$.

[3] Or a finite set of its minimal elements if $\succeq$ is not a partial order.

[4] A function that returns all the one-step successors states of a given set of states.

and not a generic symbolic representation of such sets. As a consequence, their theory has to be instantiated for the particular class of WSTS that is targeted and this is not a trivial task. Second, as downward closed sets are abstractions for sets of reachable states in the forward algorithm, we formalize our generic representation of downward closed set as a generic abstract domain. This allow us to rephrase in a simpler way the forward algorithm, first proposed in [14], in the context of abstract interpretation. Third, we show how to automatically refine the abstract domain in order to obtain, in an efficient way, overapproximations that are guaranteed to be sufficiently precise to decide the coverability problem.

Our paper is organized as follows. Section 2 presents some preliminaries. Section 3 introduces the generic representation of downward closed sets. In Section 4 we will be concerned with the abstract interpretation of WSTS. Section 5 is devoted to the refinement of the abstract domain. Section 6 shows on an example how these techniques work. A version of the paper containing all proofs is available at [16].

2 Preliminaries

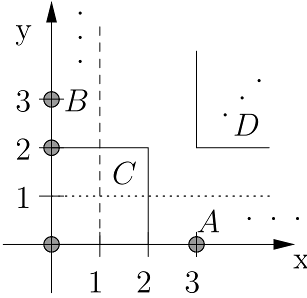
2.1 Well-Quasi Ordered Sets

A *preorder* $\succeq$ is a binary relation over a set X which is reflexive, and transitive. The preorder $\succeq$ is a *well-quasi order* (wqo for short) if there is no infinite sequence $x_0, x_1, \ldots$, such that $x_i \not\succeq x_j$ for all $i > j \geq 0$. A set $M \subseteq X$ is said to be *canonical* if for any distinct $x, y \in M$ we have $x \not\succeq y$. We say that $M \subseteq S$ is a *minor set* of $S \subseteq X$, if for all $x \in S$ there exists $y \in M$ such that $x \succeq y$, and M is canonical.

Lemma 1 (From [17]). *Let $(X, \succeq)$ be a well-quasi ordered set (wqo-set for short). For any set $S \subseteq X$, S has at least one finite minor set M.*

We use min to denote a function which, given a set $S \subseteq X$, returns a minor set of S. Let $(X, \succeq)$ be a wqo-set, we call $x{\downarrow} = \{x' \in X \mid x \succeq x'\}$ and $x{\uparrow} = \{x' \in X \mid x' \succeq x\}$ the $\succeq$*-downward closure* and $\succeq$*-upward closure* of $x \in X$, respectively. This definition is naturally extended to sets in X. We define a set $S \subseteq X$ to be a $\succeq$-downward closed set ($\succeq$*-dc-set* for short), respectively $\succeq$-upward closed set ($\succeq$*-uc-set* for short), iff $S{\downarrow} = S$, respectively $S{\uparrow} = S$. Examples of such sets are given in Fig. 1. For any wqo-set $(X, \succeq)$, we define $DCS(X)$ ($UCS(X)$) to be the set of all $\succeq$-dc-sets ($\succeq$-uc-sets) in X. For any $x \in X$ we define the $\succeq$-equivalence class of x, denoted $[x]$, to be the set $x{\uparrow} \cap x{\downarrow}$, i.e. the set of elements that are $\succeq$-equivalent to x. For A and B subsets of X, we say that $A \equiv B$ if $A{\uparrow} = B{\uparrow}$. Observe that $A \equiv B$ iff for all $a \in A$ there is a $b \in B$ such that $a \succeq b$, and vice versa. We now recall a well-known lemma on $\succeq$-uc-sets and $\succeq$-dc-sets.

Lemma 2 (From [17]). *Let $(X, \succeq)$ a wqo-set and an infinite sequence of $\succeq$-uc-set $U_0 U_1 \ldots$ such that $\forall i \geq 0\colon U_i \subseteq U_{i+1}$. There exists $j \geq 0\colon \forall j' \geq j\colon U_j = U_j$. Symmetrically, given an infinite sequence of $\succeq$-dc-sets $D_0 D_1 \ldots$ such that $\forall i \geq 0\colon D_i \supseteq D_{i+1}$, there exists $j \geq 0\colon \forall j' \geq j\colon D_j = D_j$.*



The wqo $\succeq$ is defined as follows $(a_1, a_2) \succeq (b_1, b_2)$ if and only if $a_1 \geq b_1$ and $a_2 \geq b_2$. The $\succeq$-dc-sets A and B are infinite $\succeq$-dc-set : $A = \{(x, y) \in \mathbb{N}^2 \mid y \leq 1\}$, $B = \{(x, y) \in \mathbb{N}^2 \mid x \leq 1\}$. On the contrary, the $\succeq$-dc-set $C = \{(x, y) \in \mathbb{N}^2 \mid x \leq 2 \wedge y \leq 2\}$ is finite. The $\succeq$-uc-set D is given by $\{(x, y) \in \mathbb{N}^2 \mid x \geq 3 \wedge y \geq 2\}$. Note that D has exactly one minor set since $\succeq$ is a partial order.

Fig. 1. $\succeq$-dc-sets and $\succeq$-uc-sets in $\mathbb{N}^2$

We now introduce a lemma stating several facts about sets and their closure. These facts are merely of technical interest and will be used subsequently.

Lemma 3.

1. *For any* $S, S' \subseteq X$, $S{\downarrow} \cap S'{\uparrow} \neq \emptyset \Leftrightarrow S{\downarrow} \cap S' \neq \emptyset \Leftrightarrow S \cap S'{\uparrow} \neq \emptyset$.
2. *For any* $S, S' \subseteq X$, $S{\uparrow} \subseteq S'{\uparrow} \Leftrightarrow \forall s \in S\ \exists s' \in S' : s \succeq s'$.
3. $\forall s \in X, S \in UCS(X) : s \in S \Leftrightarrow \exists s' \in \mathsf{min}(S) : s \succeq s'$.

Lemma 3.2 and 3.3 suggest an effective representation of $\succeq$-uc-sets: every $\succeq$-uc-set U can be finitely represented by $\mathsf{min}(U)$. For decidable well-quasi order $\succeq$, this readily gives us an effective procedure to check inclusion between two $\succeq$-uc-sets, to check membership and to compute union [18].

Notations. Sometimes we write s instead of the set $\{s\}$. Unless otherwise stated the transitive and reflexive closure f^* of a function f such that its domain and co-domain coincide is given by $\bigcup_{i \geq 0} f^i$ where f^0 is the identity and $f^{i+1} = f^i \circ f$. Finally, let us recall the following property on sets that we will use without mention in our proofs: $A \subseteq B$ iff $A \cap (X \setminus B) = \emptyset$.

2.2 Well-Structured Transitions Systems

In this paper we follow [19] in the definition of well-structured transition systems.

Definition 1. *A* well-structured transition system *(*WSTS*)* $\mathcal{S}$ *is a tuple* $(X, \delta, \succeq)$ *where* X *is a (possibly) infinite set of states,* $\delta \subseteq X \times X$ *is a transition relation between states — we use the notation* $x \to x'$ *if* $(x, x') \in \delta$ *—, and* $\succeq \subseteq X \times X$ *is a preorder between states such that the two following conditions hold:* (i) $\succeq$ *is a wqo; and* (ii) $\forall x_1, x_2, x_3\ \exists x_4 : (x_3 \succeq x_1 \wedge x_1 \to x_2) \Rightarrow (x_3 \to^* x_4 \wedge x_4 \succeq x_2)$, *where* $\to^*$ *is the reflexive and transitive closure of the transition relation (upward compatibility)*[5]*. Moreover, we define an* initialized WSTS *(*IWSTS*) to be a pair* $(\mathcal{S}, x_0)$ *where* $\mathcal{S} = (X, \delta, \succeq)$ *is a* WSTS *and* $x_0 \in X$ *is the* initial state. *We adhere to the convention that if* $\mathcal{S}_0$ *is an* IWSTS *then* $\mathcal{S}$ *is its* WSTS.

[5] Upward compatibility is more general than the compatibility used in [17].

Let $\mathcal{S} = (X, \delta, \succeq)$ be a WSTS and $T \subseteq X$, $\mathit{post}[\mathcal{S}](T) \stackrel{\text{def}}{=} \{x' \mid \exists x \in T \colon x \to x'\}$. Analogously, we define $\mathit{pre}[\mathcal{S}](T)$ as $\{x \mid \exists x' \in T \colon x \to x'\}$. We define $\mathit{minpre}[\mathcal{S}](T) \stackrel{\text{def}}{=} \mathsf{min}((\mathit{pre}[\mathcal{S}](T{\uparrow})){\uparrow})$. To shorten notation, we write pre,minpre and post if the WSTS is clear from the context. The following definition follows [17].

Definition 2. *An* effective WSTS *is a* WSTS $\mathcal{S} = (X, \delta, \succeq)$ *where both* $\succeq$ *and* $\to$ *are decidable and for all* $x \in X$: $\mathit{minpre}[\mathcal{S}](x)$ *is computable.*

2.3 The Coverability Problem

The verification of safety properties on IWSTS reduces to the so called coverability problem.

Problem 1. The *coverability problem* for IWSTS is defined as follows: "*Given an* IWSTS $((X, \delta, \succeq), x_0)$ *and* $\mathsf{bad} \in \mathit{UCS}(X)$, $\mathit{post}^*(x_0) \cap \mathsf{bad} = \emptyset$*?* "

In general, bad is an upward closed set of states where errors occur.

Two solutions to the coverability problem can be found in the literature. The first one (see [17, 19]) is a backward approach based on the following two lemmas:

Lemma 4 (From [19]). *Given a* WSTS $\mathcal{S} = (X, \delta, \succeq)$ *and* $U \in \mathit{UCS}(X)$, *(a)* $\mathit{pre}^*(U) \in \mathit{UCS}(X)$, *and (b)* $\mathit{minpre}^*(\mathsf{min}(U)){\uparrow} = \mathit{pre}^*(U)$.

Lemma 4, together with Lemma 1 and 2, show how to (symbolically) compute the (possibly) infinite set $\mathit{pre}^*(U)$ using the minor sets of $\succeq$-uc-sets. Once $\mathit{pre}^*(U)$ is computed, or rather a finite representation using one of its minor set, one can decide the coverability problem by testing if the initial state is in $\mathit{pre}^*(U)$ by using Lemma 3.3.

The second approach is a forward approach based on the notion of *covering set* [20, 12]. The covering set $\mathit{Cover}(\mathcal{S}_0)$ of an IWSTS $\mathcal{S}_0 = (\mathcal{S}, x_0)$ is given by $\mathit{Cover}(\mathcal{S}_0) \stackrel{\text{def}}{=} \mathit{post}^*(x_0){\downarrow}$. The following lemma shows the usefulness of covering sets to solve the coverability problem:

Lemma 5. *Given an* IWSTS $\mathcal{S}_0 = ((X, \delta, \succeq), x_0)$ *and* $\mathsf{bad} \in \mathit{UCS}(X)$, $\mathit{Cover}(\mathcal{S}_0) \cap \mathsf{bad} = \emptyset$ *if and only if* $\mathit{post}^*(x_0) \cap \mathsf{bad} = \emptyset$.

As already mentioned in the introduction, there are two difficulties to overcome when trying to design a forward algorithm for the coverability problem:

1. Currently, there are no generic way to effectively represent and manipulate $\succeq$-dc-sets (as the one shown above for $\succeq$-uc-sets). So, for every wqo-set $(X, \succeq)$ one has to design a symbolic representation for the sets in $\mathit{DCS}(X)$.
2. The set $\mathit{Cover}(\mathcal{S}_0)$ is in general not effectively constructible, see [10] for details. As a consequence, all the algorithms based on its construction (except the well-known Karp-Miller algorithm on Petri nets) may fail to terminate.

To overcome those two difficulties:

1. In [15], the authors propose a methodology to design a symbolic representation of dc-sets. However the design of such a symbolic data-structure is far from being trivial.
2. The authors of this paper proposed, in [14], an algorithmic schema called *expand, enlarge and check* which can be instantiated for any class of WSTS as long as a symbolic representation of dc-sets is provided (called there an adequate set of limits).

In this paper, we provide, in our opinion, a much more satisfactory answer to those two difficulties by providing, in the form of a generic abstract domain and a generic abstract analysis, a completely generic algorithm to solve the coverability problem for WSTS.

3 A Generic Abstract Domain

In this section, we present a parametrized abstract domain that allows us to represent any $\succeq$-dc-set in a wqo-set $(X, \succeq)$. The parameter D is a finite subset of X and it defines the precision of the abstract domain. We also show that this parametrized abstract domain enjoys the following properties: (i) our parametrized abstract domain defines a complete lattice, (ii) we define an abstraction and a concretisation function that is shown to be a Galois insertion, (iii) any $\succeq$-dc-set can be exactly represented by our parametrized abstract domain provided an adequate value for the parameter D is used, and (iv) each $\succeq$-dc-set has a finite representation.

Recall that the powerset lattice $PL(A)$ associated to a set A is the complete lattice having the powerset of A as carrier, and union and intersection as least upper bound and greatest lower bound, respectively. In our setting the concrete lattice is the powerset lattice $PL(X)$ of the set of states X.

Fix a finite set $D \subseteq X$ which is called the *finite domain*, the abstract lattice $DPL(D)$ has $DCS(D)$ as a carrier, $\sqcup_D$ as the *least upper bound* operator, $\sqcap_D$ as the *greatest lower bound* operator, and D and $\emptyset$ are the $\sqsubseteq_D$-maximal and $\sqsubseteq_D$-minimal element, respectively. We define the relation $\sqsubseteq_D$ over $DCS(D) \times DCS(D)$ such that for all $P_1, P_2 \in DCS(D)$: $P_1 \sqsubseteq_D P_2$ if and only if $P_1 \subseteq P_2$, $P_1 \sqcup_D P_2 \stackrel{\text{def}}{=} P_1 \cup P_2$, $P_1 \sqcap_D P_2 \stackrel{\text{def}}{=} P_1 \cap P_2$. Notice that $DPL(D)$ is complete because the union and the intersection operations are closed in $DPL(D)$. Given an abstract lattice $DPL(D)$, the abstraction and concretisation mappings are given as follows:

$$\forall E \in PL(X): \qquad \alpha[D](E) \stackrel{\text{def}}{=} E\!\downarrow \cap D$$
$$\forall P \in DPL(D): \qquad \gamma[D](P) \stackrel{\text{def}}{=} \{x \in X \mid x\!\downarrow \cap D \subseteq P\} \ .$$

The set between brackets defines the *parameter* of the function and the set between parentheses is its *argument*. For simplicity of notation, we also write $\gamma(P)$, $\alpha(E)$, $\sqsubseteq$, $\sqcup$ and $\sqcap$ if the parameter is clear from the context.

We next show through an example that the finite domain D actually parametrizes the precision of the abstract domain with respect to the concrete domain.

Example 1. Let us consider the $\succeq$-dc-sets of Fig. 1 and consider the following finite domain $D = \{(0,0),(3,0),(0,2),(0,3)\}$ depicted by the grey dots. Applying α on the $\succeq$-dc-sets A, B and C give, respectively, the (abstract) sets $\alpha(A) = \{(0,0),(3,0)\}, \alpha(B) = \{(0,0),(0,2),(0,3)\}, \alpha(C) = \{(0,0),(0,2)\}$. A and C are exactly represented, i.e. $\gamma(\alpha(A)) = A$ and $\gamma(\alpha(C)) = C$, but B is not: $\gamma(\alpha(B)) = \{(x,y) \in \mathbb{N}^2 \mid x \leq 2\}$. But, if we add $(2,0)$ to D then B becomes representable.

This generic abstract domain is a generalization of the ideas exposed in [21] for finite states systems.

Fix a finite domain D, the *concrete* $PL(X)$ and *abstract* $DPL(D)$ domains and the *abstraction* $\alpha\colon PL(X) \mapsto DPL(D)$ and *concretisation* $\gamma\colon DPL(D) \mapsto PL(X)$ maps form a *Galois insertion*, denoted by $PL(X) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} DPL(D)$.

Proposition 1. *For every finite domain D, $PL(X) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} DPL(D)$.*

Proof. Fix a finite domain D. It follows immediately from the definitions that α is monotonic (i.e., $C \subseteq C'$ implies $\alpha(C) \sqsubseteq \alpha(C')$) and γ as well. Indeed, $\gamma(P_1) \subseteq \gamma(P_2) \Leftrightarrow \{c \mid c{\downarrow} \cap D \subseteq P_1\} \subseteq \{c \mid c{\downarrow} \cap D \subseteq P_2\} \Leftrightarrow P_1 \subseteq P_2 \Leftrightarrow P_1 \sqsubseteq P_2$. So, it suffices to prove (a) and (b) below:

(a) $C \subseteq (\gamma \circ \alpha)(C)$ for every $C \in PL(X)$.

$$\begin{aligned}(\gamma \circ \alpha)(C) &= \{c \in X \mid c{\downarrow} \cap D \subseteq C{\downarrow} \cap D\} \\ &\supseteq \{c \in C \mid c{\downarrow} \cap D \subseteq C{\downarrow} \cap D\} \\ &= C\end{aligned}$$

(b) $(\alpha \circ \gamma)(P) = P$ for every $P \in DPL(D)$.

$$\begin{aligned}(\alpha \circ \gamma)(P) &= \{c \mid c{\downarrow} \cap D \subseteq P\}{\downarrow} \cap D \\ &= \{c \mid c{\downarrow} \cap D \subseteq P\} \cap D && \gamma(P){\downarrow} = \gamma(P) \\ &= \{c \in D \mid c{\downarrow} \cap D \subseteq P\} \\ &= P && P \subseteq D \text{ and } P \in DCS(D)\end{aligned}$$

□

We now prove some properties on the precision of our abstract domain. The next lemma states that any $\succeq$-dc-set of X can be represented exactly using a finite domain D and a set $P \in DCS(D)$.

Lemma 6 (Completeness of the abstract domain). *For each $E \in DCS(X)$ there exists a finite domain D such that $(\gamma \circ \alpha)(E) = E$.*

Proof. Given E, we define the finite domain D to be $D = \mathsf{min}(X \setminus E)$. We prove $(\gamma \circ \alpha)(E) = E$.

Let us show that $(\gamma \circ \alpha)(E) \subseteq E$. For that, suppose by contradiction that there exists $p \in (\gamma \circ \alpha)(E) \wedge p \notin E$.

$$
\begin{aligned}
& p \notin E \\
& \Leftrightarrow p{\downarrow} \not\subseteq E && E \in DCS(X) \\
& \Leftrightarrow p{\downarrow} \cap (X \setminus E) \neq \emptyset \\
& \Leftrightarrow p{\downarrow} \cap \mathsf{min}(X \setminus E) \neq \emptyset && \text{Lem. 3.1} \\
& \Leftrightarrow \exists p' \colon p' \in p{\downarrow} \wedge p' \in \mathsf{min}(X \setminus E) \\
& \Rightarrow \exists p' \colon p' \in p{\downarrow} \wedge p' \in \mathsf{min}(X \setminus E) \wedge \exists p'' \in [p'] \colon p'' \in D && \text{def. of } D \\
& \Leftrightarrow \exists p'' \colon p'' \in p{\downarrow} \wedge p'' \in D \wedge p'' \notin E && p, p' \succeq p''; p'{\downarrow} \cap E = \emptyset
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
& p \in (\gamma \circ \alpha)(E) \\
& \Leftrightarrow p{\downarrow} \cap D \subseteq \alpha(E) && \text{def. of } \gamma \\
& \Leftrightarrow p{\downarrow} \cap D \subseteq E{\downarrow} \cap D && \text{def. of } \alpha \\
& \Leftrightarrow p{\downarrow} \cap D \subseteq E \cap D && E \in DCS(X) \\
& \Leftrightarrow p{\downarrow} \cap D \subseteq E \\
& \Leftrightarrow \forall p' \colon p' \in p{\downarrow} \wedge p' \in D \Rightarrow p' \in E \\
& \Leftrightarrow \neg\neg \left(\forall p' \colon p' \in p{\downarrow} \wedge p' \in D \Rightarrow p' \in E\right) \\
& \Leftrightarrow \neg \left(\exists p' \colon p' \in p{\downarrow} \wedge p' \in D \wedge p' \notin E\right)
\end{aligned} \tag{2}
$$

From (1) and (2) follows a contradiction.

$E \subseteq (\gamma \circ \alpha)(E)$ is immediate by property of Galois insertion. So, we have proved that $(\gamma \circ \alpha)(E) = E$. □

Remark 1. While previous lemma states that any $\succeq$-dc-set can be represented using an adequate finite domain D, there is usually no finite domain D which is able to represent all the $\succeq$-dc-sets. It should be pointed out that $\succeq$-dc-sets can be easily represented through their ($\succeq$-uc-set) complement, i.e. by using a finite set of minimal elements of their complement. However with this approach the manipulation of $\succeq$-dc-sets is not obvious. In particular, there is no generic way to compute the *post* operation applied on a $\succeq$-dc-set by manipulating its complement. Also, as $Cover(\mathcal{S}_0)$ is not constructible, it is, in some sense, useless to try to represent exactly the $\succeq$-dc-sets encountered during the forward exploration. On the other hand, we will see in Section 4 that our abstract domain allow us to define an effective and generic abstract post operator.

Hereunder, Proposition 3 shows that the more elements you put into the finite domain D, the more $\succeq$-dc-sets the abstract domain is able to represent exactly. Proposition 2, which is used in many proofs, provides an equivalent definition for $\gamma[D](P)$.

Proposition 2. *Fix a finite domain D, for every $P \in DPL(D)$ we have $\gamma(P) = X \setminus (D \setminus P){\uparrow}$.*

Proposition 3. *Fix two finite domains D and D' such that $D \subset D'$. For every $P \in DPL(D)$, there exists a $P' \in DPL(D')$ such that $\gamma[D](P) = \gamma[D'](P')$.*

Effectiveness. It is worth pointing that since we impose finiteness of D then $\sqcup_D$, $\sqcap_D$ are effective and $\sqsubseteq_D$ is decidable. So, given a finite domain D, the complete lattice $DPL(D)$ represents an effective way to manipulate (infinite) $\succeq$-dc-sets.

Even if D is finite, it can be very large and so the abstract domain may be computationally expensive to manipulate. Compact data structures like Binary Decision Diagrams [22] and Sharing Trees [23, 18] may be necessary to use in practice.

In Sect. 5 we need to decide the intersection emptiness between an $\succeq$-uc-set and a $\succeq$-dc-set. In input of this problem we are given an effective representation of these two sets. Then we solve the problem using the result of Lemma 3.1 together with the following proposition.

Proposition 4. *Fix a finite domain D, for all $P \in DPL(D)$ there exists an effective procedure to answer the membership test, i.e. "given $c \in X$, does c belong to $\gamma(P)$?".*

4 Abstract Interpretation

In this section, we define the forward abstract interpretation of a WSTS using an abstract domain parametrized by D as defined in the previous section.

Let $\mathcal{S}$ be a WSTS and D be a finite domain, $post^{\#}[\mathcal{S}, D]\colon DPL(D) \mapsto DPL(D)$ is the function defined as follows: $post^{\#}[\mathcal{S}, D] \stackrel{\text{def}}{=} \lambda P.(\alpha[D] \circ post[\mathcal{S}] \circ \gamma[D])(P)$. The function $post^{\#}[\mathcal{S}, D]^*\colon DPL(D) \mapsto DPL(D)$ is defined as follows: $post^{\#}[\mathcal{S}, D]^* \stackrel{\text{def}}{=} \lambda P. \sqcup_{i \geq 0} post^{\#}[\mathcal{S}, D]^i(P)$. We shorten $post^{\#}[\mathcal{S}, D]$ to $post^{\#}$ and $post^{\#}[\mathcal{S}, D]^*$ to $(post^{\#})^*$ if the WSTS and the finite domain are clear from the context.

The following lemma establishes the soundness of our abstract interpretation of WSTS which follows by property of Galois connection:

Lemma 7. *Given a WSTS $(X, \delta, \succeq)$ with $I \subseteq X$ and a finite domain D, (i) $post(I) \subseteq (\gamma \circ post^{\#} \circ \alpha)(I)$ and (ii) $post^*(I) \subseteq (\gamma \circ (post^{\#})^* \circ \alpha)(I)$.*

The next proposition shows that we can improve the precision of the analysis by improving the precision of the abstract domain.

Proposition 5 ($post^{\#}$ Monotonicity). *Given a WSTS $\mathcal{S} = (X, \delta, \succeq)$, two finite domains D, D' with $D \subseteq D'$, and two sets $C, C' \subseteq X$ with $C \subseteq C'$, we have,* (1) $(\gamma[D'] \circ post^{\#}[\mathcal{S}, D'] \circ \alpha[D'])(C) \subseteq (\gamma[D] \circ post^{\#}[\mathcal{S}, D] \circ \alpha[D])(C')$*; and* (2) $(\gamma[D'] \circ post^{\#}[\mathcal{S}, D']^* \circ \alpha[D'])(C) \subseteq (\gamma[D] \circ post^{\#}[\mathcal{S}, D]^* \circ \alpha[D])(C')$.

Let us now show that if we fix a finite domain D, then $post^{\#}$ is computable for any effective WSTS but first we need the following lemma:

Lemma 8. *Given a WSTS $\mathcal{S}=(X, \delta, \succeq)$, $\forall x, x' \in X : x \in pre(x'\uparrow) \Leftrightarrow x' \in post(x)\downarrow$.*

Proof. $x \in pre(x'\uparrow) \Leftrightarrow \exists x''\colon x'' \succeq x' \wedge x \to x'' \Leftrightarrow x' \in post(x)\downarrow$. □

We have the following characterization of $post^{\#}$.

Proposition 6. *Fix a finite domain D, and an effective WSTS $\mathcal{S} = (X, \delta, \succeq)$. For every $x \in D$ and $P \in DPL(D)$:*

$$x \in post^{\#}(P) \Leftrightarrow (x \in D \wedge \neg(pre(x\uparrow) \subseteq (D \setminus P)\uparrow)) \quad .$$

The sets $pre(x\uparrow)\uparrow$ and $(D \setminus P)\uparrow$ are $\succeq$-uc-sets which have as finite minor set $minpre(x)$ and $(D \setminus P)$ respectively. Lemma 3.2 shows that if both $minpre(x)$ and

$(D \setminus P)$ are finite sets and $\succeq$ is decidable then we have an *effective* procedure to decide if $pre(x\uparrow)\uparrow \subseteq (D \setminus P)\uparrow$ which is equivalent to $pre(x\uparrow) \subseteq (D \setminus P)\uparrow$. Furthermore, since the complete lattice $DPL(D)$ is finite, it follows that:

Corollary 1. *For any effective* IWSTS $\mathcal{S}_0 = (\mathcal{S}, x_0)$*, and any finite domain* D*,* $((post^{\#}[\mathcal{S}, D])^* \circ \alpha)(x_0)$ *can be effectively computed.*

5 Domain Refinements

In this section, we show that the abstract interpretation that we have defined previously can be made sufficiently precise to decide the coverability problem of (effective) IWSTS. We present two ways of achieving completeness of the abstract interpretation. Both are based on abstract domain refinement. The first (and naïve) way is through enumeration of finite domains. The enumerating algorithm shows that completeness is achievable by systematically enlarging the finite domain D. The second algorithm, which is more sophisticated, enlarges the finite domain D using abstract counter-examples.

5.1 Enumerate Finite Domains

In Sect. 3, we showed that any $\succeq$-dc-set can be represented using a well chosen domain (Lemma 6). In particular, the covering set can be represented using a finite domain D.

Hereunder, Theorem 1 asserts that the abstract interpretation of an IWSTS $\mathcal{S}_0$ using a finite domain D that allows to represent exactly the covering set of $\mathcal{S}_0$ leads to the construction of that set.

Theorem 1. *Given* $Cover(\mathcal{S}_0)$*, the covering set of an* IWSTS $\mathcal{S}_0$*, and some finite domain* D *such that there is* $\Theta \in DPL(D)$: $\gamma(\Theta) = Cover(\mathcal{S}_0)$*. For any* $P \in DPL(D)$ *such that* $P \sqsubseteq \Theta$ *we have* $(\gamma \circ (post^{\#})^*)(P) \subseteq Cover(\mathcal{S}_0)$*.*

Proof.

$$
\begin{aligned}
&\gamma(\Theta) = Cover(\mathcal{S}_0) && \text{by hypothesis}\\
&\Rightarrow (post \circ \gamma)(\Theta) = post(Cover(\mathcal{S}_0)) && \text{monotonicity of } post\\
&\Rightarrow (post \circ \gamma)(\Theta) \subseteq Cover(\mathcal{S}_0) && post(Cover(\mathcal{S}_0)) \subseteq Cover(\mathcal{S}_0)\\
&\Rightarrow (\alpha \circ post \circ \gamma)(\Theta) \sqsubseteq \alpha(Cover(\mathcal{S}_0)) && \text{by monotonicity of } \alpha\\
&\Leftrightarrow post^{\#}(\Theta) \sqsubseteq \alpha(Cover(\mathcal{S}_0)) && \text{def. of } post^{\#}\\
&\Leftrightarrow post^{\#}(\Theta) \sqsubseteq \Theta && \gamma(\Theta) = Cover(\mathcal{S}_0), \Theta = (\alpha \circ \gamma)(\Theta) \quad (3)
\end{aligned}
$$

Since $post^{\#}$ is a monotone function on a complete lattice, (3) shows that for any $P \sqsubseteq \Theta$ we have

$$
\begin{aligned}
&((post^{\#})^*)(P) \sqsubseteq \Theta\\
&\Rightarrow (\gamma \circ (post^{\#})^*)(P) \subseteq \gamma(\Theta) && \text{monotonicity of } \gamma\\
&\Leftrightarrow (\gamma \circ (post^{\#})^*)(P) \subseteq Cover(\mathcal{S}_0) && \text{by hypothesis}
\end{aligned}
$$

□

Thanks to this proposition and the results of [14] Algorithm 1 decides the coverability problem for an effective IWSTS $\mathcal{S}_0 = (\mathcal{S}, x_0)$ and a $\succeq$-uc-set bad. The main idea underlying the algorithm is to iteratively analyze an underapproximation of the reachable states (line 1) followed by an overapproximation (line 2). Positive instances of the coverability problem are decided by underapproximations and negative instances are decided by overapproximations. By enumeration of finite domains D_i and Theorem 1, it is ensured that our abstract interpretation will eventually become precise enough for the negative instances. For this algorithm

Algorithm 1. Enumeration

```
  Input: An IWSTS S_0 = ((X, δ, ⪰), x_0) and a set bad ∈ UCS(X)
  for D_i = D_0, D_1, ... an enumeration of the finite subsets of X do
1     if ∃x_0, ..., x_k ∈ D_i : x_0 → ... → x_k ∧ x_k ∈ bad then
          return REACHABLE
2     else if (γ[D_i] ∘ (post#[S, D_i])* ∘ α[D_i])(x_0) ∩ bad = ∅ then
          return UNREACHABLE
  end
```

to be effective, we only need the (mild) additional assumption that elements of X are enumerable.

In the next subsection, we show that this assumption can be dropped and propose a more sophisticated way to obtain a finite domain D which is precise enough to solve the coverability problem. Our refinement technique is based on the analysis of the states leading to bad.

5.2 Eliminate Overapproximations Leading to bad

Let us first consider the following lemma that is a first step towards completeness.

Lemma 9. *Given a* WSTS $(X, \delta, \succeq)$ *and a set* $\mathsf{bad} \in UCS(X)$ *fix a finite domain* D *and a set* $P' \in DPL(D)$ *such that* $post^{\#}(P') \sqsubseteq P'$ *and* $\min(pre^*(\mathsf{bad})) \cap \gamma(P') \subseteq D$. *For every* $P \in DPL(D)$ *such that* $P \sqsubseteq P'$ *we obtain* $\gamma(P) \cap pre^*(\mathsf{bad}) = \emptyset \Rightarrow \gamma(post^{\#}(P)) \cap pre^*(\mathsf{bad}) = \emptyset$.

Proof.

$$\begin{aligned}
&\gamma(P) \cap pre^*(\mathsf{bad}) = \emptyset \\
&\Leftrightarrow (\downarrow \circ post \circ \gamma)(P) \cap pre^*(\mathsf{bad}) = \emptyset && \text{Lem. 8 and } pre(pre^*(\mathsf{bad})) = pre^*(\mathsf{bad}) \\
&\Rightarrow (\alpha \circ post \circ \gamma)(P) \cap pre^*(\mathsf{bad}) = \emptyset && (\alpha \circ post \circ \gamma)(P) \subseteq (\downarrow \circ post \circ \gamma)(P) \\
&\Leftrightarrow post^{\#}(P) \cap pre^*(\mathsf{bad}) = \emptyset && \text{def. of } post^{\#} \\
&\Leftrightarrow pre^*(\mathsf{bad}) \subseteq (X \setminus post^{\#}(P))
\end{aligned}$$

So we have established

$$\gamma(P) \cap pre^*(\mathsf{bad}) = \emptyset \Rightarrow pre^*(\mathsf{bad}) \subseteq (X \setminus post^{\#}(P)) \ . \tag{4}$$

Moreover, we conclude from $P \sqsubseteq P'$ that $post^{\#}(P) \sqsubseteq post^{\#}(P')$ (by monotonicity of $post^{\#}$), hence that $post^{\#}(P) \sqsubseteq P'$ ($post^{\#}(P') \sqsubseteq P'$) and finally that $\gamma(post^{\#}(P)) \subseteq \gamma(P')$ (by monotonicity of γ).

Now, let us consider $\gamma(post^{\#}(P))$:

$$
\begin{array}{ll}
\gamma(post^{\#}(P)) & \\
= \{c \mid c{\downarrow} \cap D \subseteq post^{\#}(P)\} & \text{definition of } \gamma \\
= \{c \in \gamma(P') \mid c{\downarrow} \cap D \subseteq post^{\#}(P)\} & \gamma(post^{\#}(P)) \subseteq \gamma(P') \\
\subseteq \{c \in \gamma(P') \mid c{\downarrow} \cap \mathsf{min}(pre^*(\mathsf{bad})) \cap \gamma(P') \subseteq post^{\#}(P)\} & \text{def. of } D \\
= \{c \in \gamma(P') \mid c{\downarrow} \cap \mathsf{min}(pre^*(\mathsf{bad})) \subseteq post^{\#}(P)\} & c \in \gamma(P') \text{ implies } c{\downarrow} \subseteq \gamma(P') \\
= \{c \in \gamma(P') \mid c{\downarrow} \cap \mathsf{min}(pre^*(\mathsf{bad})) \cap (X \setminus post^{\#}(P)) = \emptyset\} & \\
\subseteq \{c \in \gamma(P') \mid c{\downarrow} \cap \mathsf{min}(pre^*(\mathsf{bad})) \cap pre^*(\mathsf{bad}) = \emptyset\} & \text{By (4)} \\
= \{c \in \gamma(P') \mid c{\downarrow} \cap \mathsf{min}(pre^*(\mathsf{bad})) = \emptyset\} & \mathsf{min}(A) \subseteq A \text{ if } A \in UCS(X) \\
= \{c \in \gamma(P') \mid \{c\} \cap pre^*(\mathsf{bad}) = \emptyset\} & \text{Lem. 3.1} \\
= \{c \in \gamma(P') \mid c \notin pre^*(\mathsf{bad})\} &
\end{array}
$$

Hence, $\gamma(post^{\#}(P)) \cap pre^*(\mathsf{bad}) = \emptyset$. □

Using the previous lemma and induction we can establish the following theorem.

Theorem 2. *Given a* WSTS $(X, \delta, \succeq)$ *and a set* $\mathsf{bad} \in UCS(X)$ *fix a finite domain* D *and a set* $P' \in DPL(D)$ *such that* $post^{\#}(P') \sqsubseteq P'$ *and* $\mathsf{min}(pre^*(\mathsf{bad})) \cap \gamma(P') \subseteq D$. *For every* $I \subseteq X$ *such that* $\alpha(I) \sqsubseteq P'$, *we have* $I \cap pre^*(\mathsf{bad}) = \emptyset \Leftrightarrow (\gamma \circ (post^{\#})^* \circ \alpha)(I) \cap \mathsf{bad} = \emptyset$.

We are nearly in position to define our refinement-based algorithm. We first define the following operator parametrized by $\mathcal{O} \subseteq X$ which is applied to a finite subset of states $T \subseteq X$: $minpre[\mathcal{S}, \mathcal{O}](T) \stackrel{\text{def}}{=} minpre[\mathcal{S}](T) \cap \mathcal{O}$.We also write $minpre[\mathcal{O}](T)$ instead of $minpre[\mathcal{S}, \mathcal{O}](T)$ if the WSTS is clear from the context.

In the remainder of this section we adopt the following convention: a set A acting as the argument of *minpre* should be read as $\mathsf{min}(A)$. A direct consequence of the definition of *minpre* is the following, for any $\mathcal{O} \subseteq \mathcal{O}' \subseteq X$ and $A \subseteq X$ we have:

$$minpre[\mathcal{O}]^*(A) \subseteq minpre[\mathcal{O}']^*(A) \ . \tag{5}$$

The main ideas underlying our refinement-based algorithm (Algorithm 2) are as follows. In a first approximation, we consider a finite domain D_0 that contains a minor set of bad. With this set, we compute a first overapproximation of the reachable states of $\mathcal{S}_0$, noted $\mathcal{O}_0$. If this overapproximation is fine enough to prove that we are in presence of a negative instance of the problem then we conclude at line 2. If it is not the case, we compute R'_0 that represents all the states within $\mathcal{O}_0$ that can reach bad in one step. If this set contains x_0 then we conclude that bad is reachable. Otherwise, we refine the finite domain D_0 into D_1 to ensure at the next iteration that our overapproximation will be more precise (Prop. 5.2) and that $(\gamma[D_1] \circ post^{\#}[\mathcal{S}, D_1] \circ \alpha[D_1])(x_0))$ will not intersect

with bad. So, we have excluded **all** spurious counter-examples of length one. We then proceed with this enlarged finite domain.

Since $\mathsf{min}(pre^*(\mathsf{bad}))$ is computable, Theorem 2 intuitively shows that our algorithm terminates. We formally establish the correctness of our technique as stated in the next lemmas which prove soundness, completeness, and termination of Algorithm 2.

Algorithm 2. Refinement loop

Input: An IWSTS $\mathcal{S}_0$ and a set $\mathsf{bad} \in UCS(X)$
Let $D_0 \supseteq (\mathsf{min}(\mathsf{bad}))$
for $i = 0, 1, 2, \ldots$ **do**
1 Compute R_i defined to be $((post^{\#}[\mathcal{S}, D_i])^* \circ \alpha[D_i])(x_0)$
Let $\mathcal{O}_i$ denote $\gamma[D_i](R_i)$
2 **if** $\mathcal{O}_i \cap \mathsf{bad} = \emptyset$ **then return** UNREACHABLE
else
3 Compute R'_i defined to be $\mathsf{min}\left(\bigcup_{k=0}^{i+1} minpre[\mathcal{S}, \mathcal{O}_i]^k(\mathsf{bad})\right)$
4 **if** $\{x_0\} \cap R'_i{\uparrow} = \emptyset$ **then**
5 **choose** $D_{i+1} \supseteq D_i \cup R'_i$
else return REACHABLE
end
end

Lemma 10 (Soundness). *If Algorithm 2 says "*REACHABLE*" then we have* $post^*(x_0) \cap \mathsf{bad} \neq \emptyset$.

Proof. Let c be the value of variable i when the algorithm says "REACHABLE". $minpre[\mathcal{O}_c]^*(\mathsf{bad}){\uparrow} \subseteq minpre[X]^*(\mathsf{bad}){\uparrow} = pre^*(\mathsf{bad})$, the inclusion follows from $\mathcal{O}_c \subseteq X$, (5) and $\uparrow$ is monotonic, and the equality follows from Lemma 4.b. Since $\{x_0\} \cap pre^*(\mathsf{bad}) \neq \emptyset$ iff $post^*(x_0) \cap \mathsf{bad} \neq \emptyset$, $minpre[\mathcal{O}_c]^c(\mathsf{bad}){\uparrow} \subseteq pre^*(\mathsf{bad})$ shows that $post^*(x_0) \cap \mathsf{bad} \neq \emptyset$, by $\{x_0\} \cap minpre[\mathcal{O}_c]^c(\mathsf{bad}){\uparrow} \neq \emptyset$ (line 4). □

Lemma 11 (Completeness). *If Algorithm 2 says "*UNREACHABLE*" then we have* $post^*(x_0) \cap \mathsf{bad} = \emptyset$.

Proof. Fix a finite domain D, by Lemma 7 we have that $post^*(x_0) \subseteq (\gamma \circ (post^{\#})^* \circ \alpha)(x_0)$. Let c be the value of variable i when the algorithm says "UNREACHABLE" at line 2. We conclude from $(\gamma[D_c] \circ (post^{\#}[\mathcal{S}, D_c])^* \circ \alpha[D_c])(x_0) \cap \mathsf{bad} = \emptyset$ that $post^*(x_0) \cap \mathsf{bad} = \emptyset$ which is the desired conclusion. □

Lemma 12 (Termination). *Given an effective* IWSTS $\mathcal{S}_0$ *and* $\mathsf{bad} \in UCS(X)$, *Algorithm 2 always terminates.*

Proof. It is routine to check that each domain D_i is finite. Hence, since $\sqcup_{D_i}$ is computable because the D_i's are finite and $post^{\#}[\mathcal{S}, D_i]$ is computable following Proposition 6 (notice that $\alpha[D_i](x_0)$ is computable since $\succeq$ is assumed to be decidable), the fixpoint computation of line 1 finishes after a finite amount of time.

Suppose, contrary to our claim, that the algorithm does not terminate. Since each line is evaluated in a finite amount of time, it follows that the algorithm executes the main loop infinitely many times. From line 5, we conclude that the

algorithm considers an infinite sequence of finite domains $D_0 \subseteq D_1 \subseteq \cdots$ From Proposition 5.2, we know that $\mathcal{O}_0 \supseteq \mathcal{O}_1 \supseteq \cdots$ From Lemma 2, we conclude that there exists $i \geq 0$ such that $\mathcal{O}_i = \mathcal{O}_{i+1} = \cdots$

Let us consider the iteration i of the algorithm such that $\mathcal{O}_i = \mathcal{O}_{i+1} = \cdots$ We have the infinite sequence $R'_i\!\uparrow \subseteq R'_{i+1}\!\uparrow \subseteq \cdots$. From Lemma 2, we conclude that there exists $j \geq i$ such that $R'_j\!\uparrow = R'_{j+1}\!\uparrow = \cdots$. Hence, following line 5 of the algorithm, D contains $\mathsf{min}(\mathit{minpre}[\mathcal{O}_i]^*(\mathsf{bad}))$ (or rather D contains equivalent states to those of $\mathsf{min}(\mathit{minpre}[\mathcal{O}_i]^*(\mathsf{bad}))$) after the jth iteration.

Let us now prove that **(a)** $\mathsf{min}(\mathit{minpre}[\mathcal{O}_{j+1}]^*(\mathsf{bad})) \equiv \mathsf{min}(\mathit{minpre}[X]^*(\mathsf{bad})) \cap \mathcal{O}_{j+1}$. Indeed, if it is not the case there exist $l \geq 0, c, c' \in X$ such that $c \in \mathit{minpre}[X]^l(\mathsf{bad})$, $c' \in \mathit{minpre}[X](c)$, $c \notin \mathcal{O}_{j+1}$ and $c' \in \mathcal{O}_{j+1}$. Hence, $\mathit{post}(c') \nsubseteq \mathcal{O}_{j+1}$ since $\mathit{post}(c') \cap c\!\uparrow \neq \emptyset$ and $\mathcal{O}_{j+1}$ is a $\succeq$-dc-set. But, $\forall \overline{c} \in \mathcal{O}_{j+1} \colon \mathit{post}(\overline{c}) \subseteq \mathcal{O}_{j+1}$. From this follows a contradiction.

Moreover, **(b)** $\mathsf{min}((\mathit{minpre}[X]^*(\mathsf{bad}))\!\uparrow) \equiv \mathsf{min}(\mathit{pre}^*(\mathsf{bad}))$ holds by Lemma 4.b and by definition of $\equiv$. We conclude, following line 5 of the algorithm, that D_{j+1} contains equivalent states to those of $\mathsf{min}(\mathit{pre}^*(\mathsf{bad})) \cap \mathcal{O}_{j+1}$.

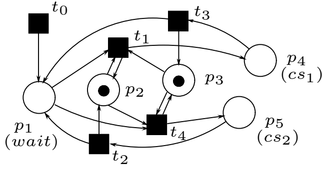
By applying Theorem 2, we have $\{x_0\} \cap \mathit{pre}^*(\mathsf{bad}) = \emptyset$ iff $\mathcal{O}_{j+1} \cap \mathsf{bad} = \emptyset$. We consider two cases: (i) $\{x_0\} \cap \mathit{pre}^*(\mathsf{bad}) = \emptyset$, then we have $\mathcal{O}_{j+1} \cap \mathsf{bad} = \emptyset$ and the algorithm terminates since the test of line 2 is evaluated to true; (ii) $\{x_0\} \cap \mathit{pre}^*(\mathsf{bad}) \neq \emptyset$, then $\mathcal{O}_{j+1} \cap \mathsf{bad} \neq \emptyset$. Following **(a)** and **(b)** at line 3 of the algorithm $R'_{j+1} \equiv \mathsf{min}(\mathit{pre}^*(\mathsf{bad})) \cap \mathcal{O}_{j+1}$. Since $\{x_0\} \cap \mathit{pre}^*(\mathsf{bad}) \neq \emptyset$, there exists, on account of Lemma 3.3, $x \in \mathsf{min}(\mathit{pre}^*(\mathsf{bad}))\colon x_0 \succeq x$. $x_0 \in \mathcal{O}_{j+1}$ and $\mathcal{O}_{j+1} \in \mathit{DCS}(X)$ shows that $x \in \mathcal{O}_{j+1}$. We conclude from $R'_{j+1} \equiv \mathsf{min}(\mathit{pre}^*(\mathsf{bad})) \cap \mathcal{O}_{j+1}$ that $[x] \cap R'_{j+1} \neq \emptyset$, hence that $\{x_0\} \cap R'_{j+1}\!\uparrow \neq \emptyset$, and finally that the test of line 4 is evaluated to false which yields the algorithm to terminate. □

Remark 2. Let us notice that the practical efficiency of Algorithm 2 depends on (i) the preciseness of the overapproximations $\mathcal{O}_i$ and (ii) the time (and space) needed to build those overapproximations. Point (i) is crucial since rough approximations will lead to the computation of $\mathsf{min}(\mathit{pre}^*(\mathsf{bad}))$, which is time and space consuming in practice [23]. Point (ii) is important because an inefficient computation of overapproximations leads to an inefficient algorithm. Hence, a trade-off between (i) and (ii) must be chosen. This problem exceeds the scope of this paper and will be addressed in future works.

To ensure termination we require, at line 5, that the finite domain is enlarged by, at least, the states of R'_i. The algorithm remains correct if we add more states.

6 Illustrations

We have produced a prototype that implements Algorithm 2. We describe in this section the execution of that prototype when applied on a toy example. The example of IWSTS $\mathcal{S}_0$ is represented through a Petri net (see [8] for details), depicted in Fig. 2, which models a very simple mutual exclusion protocol. We want to check for safety of the protocol, that is check that *there is never more than one process in the critical sections*. The markings that violates the property, denoted bad, are given by $\{\langle 0,0,0,1,1\rangle, \langle 0,0,0,0,2\rangle, \langle 0,0,0,2,0\rangle\}\!\uparrow$. It is worth



The processes (the tokens in place p_1) can access some critical section (place p_4 or p_5) provided they acquired some lock (the tokens in places p_2 and p_3). The initial marking is given by $\langle 0,1,1,0,0\rangle$. Transition t_0 spawns processes.

Fig. 2. A simple mutual exclusion protocol

pointing that we want to establish the safety for any number of processes taking part in the protocol (recall that t_0 spawns processes).

Execution of the prototype. We describe the execution of the prototype iteration by iteration. On account of remark 2, we do not take $\mathsf{min}(\mathsf{bad})$ as initial finite domain but its downward closure instead and we do not add the set R'_i to D_i at the ith iteration but its downward closure instead. Taking the $\succeq$-downward closure of the sets allows us to efficiently prove the safeness of the protocol.

Initialisation. As mentioned before, the initial value of the finite domain, which is referred as D_0, is given by $\{\langle 0,0,0,1,1\rangle, \langle 0,0,0,0,2\rangle, \langle 0,0,0,2,0\rangle\}{\downarrow}$.

Iteration 1 (i=0). After the fixpoint computation of line 1, we have $R_0 = D_0$, and so $\mathcal{O}_0 = X$. Hence the test of line 2 fails and we compute $R'_0 = \mathsf{min}(\mathsf{bad}) \cup \{\langle 1,1,1,0,1\rangle, \langle 1,1,1,1,0\rangle\}$ which corresponds to $\mathsf{min}(\mathsf{bad} \cup \mathit{pre}(\mathsf{bad}))$. Because the test of line 4 fails, we execute line 5 and we set D_1 to $R'_0{\downarrow}$.

Iteration 2 (i=1). The fixpoint computation of line 1 ends up with $R_1 = D_1$, hence $\mathcal{O}_1 = X$. Again we perform a refinement step by (i) computing $R'_1 = \mathsf{min}(\mathsf{bad}) \cup \{\langle 0,1,1,0,1\rangle, \langle 0,1,1,1,0\rangle, \langle 2,2,1,0,0\rangle, \langle 2,1,2,0,0\rangle\}$ (which corresponds to $\mathsf{min}(\mathsf{bad} \cup \mathit{pre}(\mathsf{bad}) \cup \mathit{pre}^2(\mathsf{bad})))$ and (ii) adding tuples of $R'_1{\downarrow}$ with the ones of D_1 to obtain D_2.

Iteration 3 (i=2). The fixpoint computation of line 1 finishes with a set R_2 such that the test of line 2 ($\mathcal{O}_2 \cap \mathsf{bad} = \emptyset$) succeeds and the system is proved to be safe.

Indeed $\mathcal{O}_2 = \{(p_1,p_2,p_3,p_4,p_5) \in \mathbb{N}^5 \mid (\bigwedge_{i=2}^{5} p_i \leq 1) \wedge p_4 + p_5 \leq 1 \wedge p_3 + p_4 \leq 1 \wedge p_2 + p_5 \leq 1\}$ which is equal to $\mathit{Cover}(\mathcal{S}_0)$. Since $\mathit{Cover}(\mathcal{S}_0)$ is, in general, not computable ([10]), the equality does always not hold. Notice that $\mathit{pre}^*(\mathsf{bad})$ is computed in five iterations with the classical algorithm of [17]. Hence, the forward analysis allows to drastically cut the backward search. We hope this gain will appear also on many practical examples.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–236
2. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of LICS, IEEE Computer Society Press (1996) 278–292

3. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. **127** (1996) 91–101
4. Abdulla, P., Annichini, A., Bouajjani, A.: Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In: Proceedings of TACAS. Volume 1579 of LNCS., Springer (1999) 208–222
5. Delzanno, G., Raskin, J.F., Van Begin, L.: Towards the automated verification of multithreaded java programs. In: Proceedings of TACAS. Volume 2280 of LNCS., Springer (2002) 173–187
6. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Fast acceleration of symbolic transition systems. In: Proceedings of CAV. Volume 2725 of LNCS., Springer (2003) 118–121
7. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: Proceedings of LICS, IEEE Computer Society Press (1999) 352–359
8. Reisig, W.: Petri Nets. An introduction. Springer (1986)
9. Ciardo, G.: Petri nets with marking-dependent arc multiplicity: properties and analysis. In: Proc. of ATPN. Volume 815 of LNCS., Springer (1994) 179–198
10. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In: Proceedings of ICALP. Volume 1443 of LNCS., Springer (1998) 103–115
11. Raskin, J.F., Van Begin, L.: Petri nets with non-blocking arcs are difficult to analyse. In: Proceedings of INFINITY. Volume 96 of ENTCS., Elsevier (2003)
12. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: Proc. of LICS, IEEE Computer Society Press (1998) 70–80
13. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. (3) **2** (1952) 326–336
14. Geeraerts, G., Raskin, J.F., Van Begin, L.: Expand, Enlarge and Check: new algorithms for the coverability problem of WSTS. In: Proceedings of FSTTCS. Volume 3328 of LNCS., Springer (2004) 287–298
15. Abdulla, P., Deneux, J., Mahata, P., Nylen, A.: Forward reachability analysis of timed petri nets. In: Proceedings of Formats-FTRTFT. Volume 3253 of LNCS., Springer (2004) 343–362
16. Ganty, P., Raskin, J.F., Van Begin, L.: A complete abstract interpretation framework for coverability properties of WSTS. Technical Report 2005.57, Centre Fédéré en Vérification (CFV) (2005)
17. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proceedings of LICS, IEEE Computer Society Press (1996) 313–321
18. Delzanno, G., Raskin, J.F., Begin, L.V.: Covering sharing trees: a compact data structure for parameterized verification. Software Tools for Technology Transfer (STTT) **5** (2004) 268–297
19. Finkel, A., Schnoebelen, Ph.: Well-structured transition systems everywhere! Theoretical Computer Science **256** (2001) 63–92
20. Finkel, A.: Reduction and covering of infinite reachability trees. Inf. Comput. **89** (1990) 144–179
21. Esparza, J., Ganty, P., Schwoon, S.: Locality-based abstractions. In: Proceedings of SAS. Volume 3672 of LNCS., Springer (2005) 118–134
22. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35** (1986) 677–691
23. Van Begin, L.: Efficient Verification of Counting Abstractions for Parametric Systems. PhD thesis, Université Libre de Bruxelles (2003)

Complexity Results on Branching-Time Pushdown Model Checking

Laura Bozzelli

Università di Napoli Federico II, Via Cintia, 80126 - Napoli, Italy

Abstract. The model checking problem of pushdown systems (PMC problem, for short) against standard branching temporal logics has been intensively studied in the literature. In particular, for the modal μ-calculus, the most powerful branching temporal logic used for verification, the problem is known to be EXPTIME-complete (even for a fixed formula). The problem remains EXPTIME-complete also for the logic *CTL*, which corresponds to a fragment of the alternation-free modal μ-calculus. However, the exact complexity in the size of the pushdown system (for a fixed *CTL* formula) is an open question: it lies somewhere between PSPACE and EXPTIME. To the best of our knowledge, the PMC problem for CTL^* has not been investigated so far. In this paper, we show that this problem is 2EXPTIME-complete. Moreover, we prove that the program complexity of the PMC problem against *CTL* (i.e., the complexity of the problem in terms of the size of the system) is EXPTIME-complete.

1 Introduction

Model checking is a useful method to verify automatically the correctness of a system with respect to a desired behavior, by checking whether a mathematical model of the system satisfies a formal specification of this behavior given by a formula in a suitable propositional temporal logic. There are two types of temporal logics: linear and branching. In linear temporal logics, each moment in time has a unique possible future (formulas are interpreted over linear sequences corresponding to single computations of the system), while in branching temporal logics, each moment in time may split into several possible futures (formulas are interpreted over infinite trees, which describe all the possible computations of the system). The size of an instance of a model checking problem depends on two parameters: the size of the finite formal description of the given system and the size of the formula. In practice, the formula in normally very small, while the description of the system is often very large. Therefore, the complexity of the problem in terms of the size of the system (called *program complexity*) is very important in practice. Traditionally, model checking is applied to finite-state systems, typically modelled by labelled state-transition graphs.

Recently, the investigation of model-checking techniques has been extended to infinite-state systems. An active field of research is model-checking of infinite-state sequential systems. These are systems in which each state carries a finite, but unbounded, amount of information e.g. a pushdown store. The origin of this

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 65–79, 2006.
© Springer-Verlag Berlin Heidelberg 2006

research is the result of Muller and Schupp concerning the decidability of the monadic second-order theory of *context-free systems* [MS85]. This result can be extended to *pushdown systems* [Cau96], and it implies decidability of the model checking problem for all those logics (modal μ-calculus, CTL^*, CTL, etc.) which have effective translations to the monadic second-order logic. As this general decidability result gives a non-elementary upper bound for the complexity of model checking, researchers sought decidability results of elementary complexity. Concerning pushdown systems, model checking with branching-time logics is quite hard. In particular, Walukiewicz [Wal96] has shown that model checking these systems with respect to modal μ-calculus, the most powerful branching temporal logic used for verification, is EXPTIME-complete. Even for a fixed formula in the *alternation-free* modal μ-calculus, the problem is EXPTIME-hard in the size of the pushdown system. The problem remains EXPTIME-complete also for the logic CTL [Wal00], which corresponds to a fragment of the alternation-free modal μ-calculus. However, the exact complexity in the size of the system (for a fixed CTL formula) is an open problem: it lies somewhere between PSPACE and EXPTIME [BEM97]. In [Wal00], Walukiewicz has shown that even for the simple branching-time logic EF (a fragment of CTL) the problem is quite hard since it is PSPACE-complete (even for a fixed formula). For other branching-time temporal logics such as EG, UB (which are fragments of CTL) and CTL^* (which subsumes both CTL and LTL) the problem is still open. To the best of our knowledge, the pushdown model checking problem for CTL^* has not been investigated so far.

For standard linear temporal logics, model-checking pushdown systems with LTL and the *linear-time μ-calculus* is EXPTIME-complete [BEM97]. However, the problem is polynomial in the size of the pushdown system. It follows that the problem is only slightly harder than for finite-state systems, where it is PSPACE-complete but polynomial for any fixed formula [SC85, Var88]. For optimal pushdown model–checking algorithms, see also [EHRS00, EKS03, PV04, AEM04].

In this paper we study the pushdown model checking problem (PMC problem, for short) against CTL^* and the program complexity of the PMC problem against CTL. In particular, we state the following two results:

– The PMC problem against CTL^* is 2EXPTIME-complete (and EXPTIME-complete in the size of the system).
– The program complexity of the PMC problem w.r.t. CTL is EXPTIME-complete.

In order to solve the PMC problem for CTL^* we exploit an automata-theoretic approach. In particular, we propose an exponential time reduction (in the size of the formula) to the emptiness problem of *alphabet-free alternating parity pushdown automata*. The emptiness problem for this class of automata can be solved by a construction similar to that given in [KPV02] to solve the emptiness problem for nondeterministic parity pushdown tree automata (the algorithm in [KPV02] is based on a polynomial reduction to the emptiness of two-way alternating parity finite–state tree automata, which is known to be decidable in exponential time [Var98]). 2EXPTIME-hardness is shown by a technically non-trivial reduction from the word problem for EXPSPACE–bounded alternating Turing Machines.

EXPTIME-hardness of the pushdown model checking problem against *CTL* was shown by Walukiewicz [Wal00] using a reduction from the word problem for PSPACE-bounded alternating Turing Machines. We use the basic ideas of the construction in [Wal00] in order to prove that the program complexity of the problem (i.e., assuming the *CTL* formula is fixed) is still EXPTIME-hard.

2 Preliminaries

In this section we recall syntax and semantics of CTL^* and *CTL* [EH86, CE81]. Also, we define pushdown systems and the model checking problem.

CTL* and CTL logics. The logic CTL^* is a branching–time temporal logic [EH86], where a path quantifier, E ("for some path") or A ("for all paths"), can be followed by an arbitrary linear-time formula, allowing boolean combinations and nesting, over the usual linear temporal operators X ("next"), $\mathcal{U}$ ("until"), F ("eventually"), and G ("always"). There are two types of formulas in CTL^*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. Formally, for a finite set AP of proposition names, the class of state formulas φ and the class of path formulas θ are defined by the following syntax:

$$\varphi := prop \mid \neg\varphi \mid \varphi\wedge\varphi \mid A\,\theta \mid E\,\theta$$
$$\theta := \varphi \mid \neg\theta \mid \theta\wedge\theta \mid X\theta \mid \theta\,\mathcal{U}\,\theta$$

where $prop \in AP$. The set of state formulas φ forms the language CTL^*. The other operators can be introduced as abbreviations in the usual way: for instance, $F\theta$ abbreviates $true\,\mathcal{U}\,\theta$ and $G\theta$ abbreviates $\neg F\neg\theta$.

The *Computation Tree Logic CTL* [CE81] is a restricted subset of CTL^*, obtained restricting the syntax of path formulas θ as follows: $\theta := X\varphi \mid \varphi\,\mathcal{U}\,\varphi$. This means that X and $\mathcal{U}$ must be immediately preceded by a path quantifier.

The models for the logic CTL^* are labelled graphs $\langle W, R, \mu\rangle$ where W is a countable set of vertices, $R \subseteq W \times W$ is the edge relation, and $\mu : W \to 2^{AP}$ maps each vertex $w \in W$ to the set of atomic propositions that hold in w. Such labelled graphs are called transition systems (TS, for short) here. In this context vertices are also called (global) states. For $(w, w') \in R$, we say that w' is a successor of w. A path is a (finite or infinite) sequence of vertices $\pi = w_0, w_1, \ldots$ such that $(w_i, w_{i+1}) \in R$ for any $i \geq 0$. We denote the suffix $w_i, w_{i+1}, \ldots$ of π by π^i, and the i-th vertex of π by $\pi(i)$. A *maximal* path is either an infinite path or a finite path leading to a vertex without successors.

Let $G = \langle W, R, \mu\rangle$ be an TS, $w \in W$, and π be a maximal path of G. For a state (resp., path) formula φ (resp. θ), the satisfaction relation $(G, w) \models \varphi$ (resp., $(G, \pi) \models \theta$), meaning that φ (resp., θ) holds at state w (resp., holds along π) in G, is defined by induction. The clauses for proposition letters, negation, and conjunction are standard. For the other constructs we have:

- $(G, w) \models A\,\theta$ *iff* for each maximal path π in G from w, $(G, \pi) \models \theta$;
- $(G, w) \models E\,\theta$ *iff* there exists a maximal path π from w such that $(G, \pi) \models \theta$;

- $(G, \pi) \models \varphi$ *iff* $(G, \pi(0)) \models \varphi$;
- $(G, \pi) \models X\theta$ *iff* $\pi(1)$ is defined and $(G, \pi^1) \models \theta$;
- $(G, \pi) \models \theta_1 \, \mathcal{U} \, \theta_2$ *iff* there exists $i \geq 0$ such that $(G, \pi^i) \models \theta_2$ and for all $0 \leq j < i$, we have $(G, \pi^j) \models \theta_1$.

Pushdown systems. A pushdown system (*PDS*, for short) is a tuple $\mathcal{S} = \langle AP, \Gamma, P, \Delta, L \rangle$, where AP is a finite set of proposition names, Γ is a finite stack alphabet, P is a finite set of (control) states, $\Delta \subseteq (P \times (\Gamma \cup \{\gamma_0\})) \times (P \times \Gamma^*)$ is a *finite* set of transition rules (where $\gamma_0 \notin \Gamma$ is the *stack bottom symbol*), and $L : P \times (\Gamma \cup \{\gamma_0\}) \rightarrow 2^{AP}$ is a labelling function. A *configuration* is a pair (p, α) where $p \in P$ is a control state and $\alpha \in \Gamma^* \cdot \gamma_0$ is a stack content. For each $(p, B) \in P \times (\Gamma \cup \{\gamma_0\})$, we denote by $next_{\mathcal{S}}(p, B)$ the finite set (possibly empty) of the pairs (p', β) such that $((p, B), (p', \beta)) \in \Delta$. The size $|\mathcal{S}|$ of $\mathcal{S}$ is $|P| + |\Delta|$, with $|\Delta| = \sum_{((p,B),(p\ ,\beta)) \in \Delta} |\beta|$.

The semantics of an *PDS* $\mathcal{S} = \langle AP, \Gamma, P, \Delta, L \rangle$ is described by an TS $G_{\mathcal{S}} = \langle W, R, \mu \rangle$, where W is the set of pushdown configurations, for all $(p, B \cdot \alpha) \in W$ with $B \in \Gamma \cup \{\gamma_0\}$, $\mu(p, B \cdot \alpha) = L(p, B)$, and R is defined as follows:

- $((p, B \cdot \alpha), (p', \beta)) \in R$ *iff* there is $((p, B), (p', \beta')) \in \Delta$ such that either $B \in \Gamma$ and $\beta = \beta' \cdot \alpha$, or $B = \gamma_0$ (note that $\alpha = \varepsilon$) and $\beta = \beta' \cdot \gamma_0$ (note that every transition that removes the bottom symbol γ_0 also pushes it back).

For each configuration $w \in W$, we denote by $bd_{\mathcal{S}}(w)$ the number of successors of w (note that $bd_{\mathcal{S}}(w)$ is finite).

The *pushdown model checking* problem (PMC problem, for short) against *CTL* (resp., *CTL**) is to decide, for a given *PDS* $\mathcal{S}$, an initial configuration w_0 of $\mathcal{S}$, and a *CTL* (resp., *CTL**) formula φ, whether $(G_{\mathcal{S}}, w_0) \models \varphi$.

3 Tree Automata

In order to solve the PMC problem for *CTL**, we use an automata theoretic approach; in particular, we exploit the formalisms of *Alternating Parity (finite–state) Tree automata* (*APT*, for short) [MS87, EJ91] and *Alphabet-free alternating parity pushdown automata* (*PD-APA*, for short).

Let $\mathbb{N}$ be the set of positive integers. A *tree* T is a subset of $\mathbb{N}^*$ such that if $i \cdot x \in T$ for some $i \in \mathbb{N}$ and $x \in \mathbb{N}^*$, then also $x \in T$ and for all $1 \leq j < i$, $j \cdot x \in T$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For $x \in T$, the set of *children* (or *successors*) of x (in T) is $children(T, x) = \{i \cdot x \in T \mid i \in \mathbb{N}\}$. For $x \in T$, a (full) path π of T from x is a *minimal* set $\pi \subseteq T$ such that $x \in \pi$ and for each $y \in \pi$ such that $children(T, y) \neq \emptyset$, there is exactly one node in $children(T, y)$ belonging to π. For $k \geq 1$, the (complete) k-ary tree is the tree $\{1, \ldots, k\}^*$. For an alphabet Σ, a Σ-labelled tree is a pair $\langle T, V \rangle$ where T is a tree and $V : T \rightarrow \Sigma$ maps each node of T to a symbol in Σ. Note that $\langle T, V \rangle$ corresponds to the labelled graph $G_T = \langle T, R, V \rangle$ where $(x, y) \in R$ iff $y \in children(T, x)$. If $\Sigma = 2^{AP}$, then for a given *CTL** formula φ over AP, we say that $\langle T, V \rangle$ satisfies φ if $(G_T, \varepsilon) \models \varphi$.

For a set X, let $\mathcal{B}^+(X)$ be the set of positive boolean formulas over X. Elements of X are called *atoms*. For $Y \subseteq X$ and $\psi \in \mathcal{B}^+(X)$, we say that Y satisfies ψ iff assigning **true** to all elements of Y and assigning **false** to all elements of $X \setminus Y$, makes ψ true. For $k \geq 1$, we denote by $[k]$ the set $\{1, \ldots, k\}$.

Alternating Parity (finite–state) Tree automata (*APT*). We describe *APT* over (complete) k-ary trees for a given $k \geq 1$. Formally, an *APT* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F \rangle$, where Σ is a finite input alphabet, Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+([k] \times Q)$ is a transition function, and F is a parity acceptance condition [EJ91], i.e., $F = \{F_1, \ldots, F_m\}$ is a sequence of subsets of Q, where $F_1 \subseteq F_2 \subseteq \ldots \subseteq F_m = Q$ (m is called the index of $\mathcal{A}$).

A run of $\mathcal{A}$ on a Σ-labelled k-ary tree $\langle T, V \rangle$ (where $T = [k]^*$) is a labelled tree $\langle T_r, r \rangle$ in which each node is labelled by an element of $T \times Q$. A node in T_r labelled by (x, q) describes a copy of the automaton that is in the state q and reads the node x of T. Note that many nodes of T_r can correspond to the same node of T. The labels of a node and its children (successors) have to satisfy the transition function. Formally, a run over $\langle T, V \rangle$ is a $T \times Q$-labelled tree $\langle T_r, r \rangle$ such that $r(\varepsilon) = (\varepsilon, q_0)$ and for all $y \in T_r$ with $r(y) = (x, q)$, the following holds:

– there is a (possibly empty) set $\{(h_1, q_1), \ldots, (h_n, q_n)\} \subseteq [k] \times Q$ *satisfying* $\delta(q, V(x))$ such that for each $1 \leq j \leq n$, $j \cdot y \in T_r$ and $r(j \cdot y) = (h_j \cdot x, q_j)$.

Note that several copies of the automaton may go to the same direction and that the automaton is not required to send copies to all the directions. The automaton $\mathcal{A}$ is *symmetric* if for each $(q, \sigma) \in Q \times \Sigma$, $\delta(q, \sigma)$ is a positive boolean combination of sub-formulas (called *generators*) either of the form $\bigvee_{i=1}^{i=k}(i, q')$ or of the form $\bigwedge_{i=1}^{i=k}(i, q')$ (note that q' is independent from the specific direction i). The *size* $|\mathcal{A}|$ of a symmetric *APT* $\mathcal{A}$ is $|Q| + |\delta| + |F|$ where $|\delta| = \sum_{(q,\sigma) \in Q \times \Sigma} |\delta(q, \sigma)|$ and $|\delta(q, \sigma)|$ is the length of the formula obtained from $\delta(q, \sigma)$ considering each generator occurring in $\delta(q, \sigma)$ as an atomic proposition.

For a run $\langle T_r, r \rangle$ over $\langle T, V \rangle$ and an infinite path $\pi \subseteq T_r$, let $inf_r(\pi) \subseteq Q$ be the set such that $q \in inf_r(\pi)$ iff there are infinitely many $y \in \pi$ such that $r(y) \in T \times \{q\}$. For the parity acceptance condition $F = \{F_1, \ldots, F_m\}$, π is *accepting* if there is an *even* $1 \leq i \leq m$ such that $inf_r(\pi) \cap F_i \neq \emptyset$ and for all $j < i$, $inf_r(\pi) \cap F_j = \emptyset$. A run $\langle T_r, r \rangle$ is *accepting* if all its infinite paths are accepting. The automaton $\mathcal{A}$ accepts an input tree $\langle T, V \rangle$ iff there is an accepting run of $\mathcal{A}$ over $\langle T, V \rangle$. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of Σ-labelled (complete) k-ary trees accepted by $\mathcal{A}$.

It is well-known that formulas of *CTL** can be translated to tree automata. In particular, we are interested in optimal translations to symmetric *APT*.

Lemma 1 ([KVW00]). *Given a* CTL* *formula* φ *over AP and* $k \geq 1$*, we can construct a* symmetric APT *of size* $O(2^{|\varphi|})$ *and index* $O(|\varphi|)$ *that accepts exactly the set of* 2^{AP}*-labelled complete* k*-ary trees satisfying* φ.[1]

[1] [KVW00] gives a translation from *CTL** to *Hesitant alternating tree automata* which are a special case of parity alternating tree automata.

Alphabet-free alternating parity pushdown automata (*PD-APA*). An *PD-APA* is a tuple $\mathcal{P} = \langle \Gamma, P, p_0, \alpha_0, \rho, F \rangle$, where Γ is a finite stack alphabet, P is a finite set of (control) states, $p_0 \in P$ is an initial state, $\alpha_0 \in \Gamma^* \cdot \gamma_0$ is an initial stack content, $\rho : P \times (\Gamma \cup \{\gamma_0\}) \rightarrow \mathcal{B}^+(P \times \Gamma^*)$ is a transition function, and $F = \{F_1, \ldots, F_m\}$ is a *parity* acceptance condition over P. Intuitively, when the automaton $\mathcal{P}$ is in state p and the stack contains a word $B \cdot \alpha \in \Gamma^*.\gamma_0$, then $\mathcal{P}$ chooses a (possibly empty) finite set $\{(p_1, \beta_1), \ldots, (p_n, \beta_n)\} \subseteq P \times \Gamma^*$ satisfying $\rho(p, B)$ and splits in n copies such that for each $1 \leq j \leq n$, the j-th copy moves to state p_j and updates the stack content by removing B and pushing β_j.

Formally, a run of $\mathcal{P}$ is a $P \times \Gamma^*.\gamma_0$-labelled tree $\langle T_r, r \rangle$ such that $r(\varepsilon) = (p_0, \alpha_0)$ and for all $y \in T_r$ with $r(y) = (p, B \cdot \alpha)$ and $B \in \Gamma \cup \{\gamma_0\}$, the following holds:

– there is a (possibly empty) finite set $\{(p_1, \beta_1), \ldots, (p_n, \beta_n)\} \subseteq P \times \Gamma^*$ *satisfying* $\rho(p, B)$ such that for each $1 \leq j \leq n$, $j \cdot y \in T_r$ and $r(j \cdot y) = (p_j, \beta_j \cdot \alpha)$ if $B \neq \gamma_0$, and $r(j \cdot y) = (p_j, \beta_j \cdot \gamma_0)$ otherwise (note that in this case $\alpha = \varepsilon$).

The notion of *accepting* path $\pi \subseteq T_r$ is defined as for *APT* with $inf_r(\pi)$ defined as follows: $inf_r(\pi) \subseteq P$ is the set such that $p \in inf_r(\pi)$ iff there are infinitely many $y \in \pi$ for which $r(y) \in \{p\} \times \Gamma^* \cdot \gamma_0$. A run $\langle T_r, r \rangle$ is *accepting* if every infinite path $\pi \subseteq T_r$ is accepting. The *emptiness* problem for *PD-APA* is to decide, for a given *PD-APA*, the existence of an accepting run.

For $(p, \alpha) \in P \times \Gamma^*$, the size of (p, α) is $|\alpha|$. The size $|\rho|$ of the transition function is given by $\sum_{(p,B) \in P \times (\Gamma \cup \{\gamma_0\})} |\rho(p, B)|$ where $|\rho(p, B)|$ is the sum of the sizes of the occurrences of atoms in $\rho(p, B)$.

In the following we are interested in the emptiness problem for *PD-APA*. In [KPV02], an optimal algorithm is given to solve the emptiness problem for nondeterministic parity pushdown tree automata. This algorithm is based on a polynomial reduction to the emptiness of two-way alternating parity tree automata, which is known to be decidable in exponential time [Var98]. By a similar reduction we obtain the following.

Proposition 1. *The emptiness problem for* PD-APA *with index m and transition function ρ is solvable in time exponential in $m \cdot |\rho|$.*

4 Upper Bound for *CTL**

We solve the PMC problem for CTL^* using an automata theoretic approach. We fix an $PDS\, \mathcal{S} = \langle AP, \Gamma, P, \Delta, L \rangle$, an initial configuration $w_0 = (p_0, \alpha_0)$ of $\mathcal{S}$, and a CTL^* formula φ. The unwinding of the TS $G_\mathcal{S} = \langle W, R, \mu \rangle$ from w_0 induces a W-labelled tree $\langle T_\mathcal{S}, V_\mathcal{S} \rangle$: the root of $T_\mathcal{S}$ is associated with the initial configuration w_0, and each node $x \in T_\mathcal{S}$ labelled by $w \in W$ has $bd_\mathcal{S}(w)$ successors, each associated with a successor w' of w.[2] In the following, we sometime view $\langle T_\mathcal{S}, V_\mathcal{S} \rangle$

[2] Assuming that W is ordered, there is indeed only a single such tree. Since CTL^* formulas cannot distinguish between trees obtained by different orders, we do not lose generality by considering a particular order.

as a 2^{AP}-labelled tree, taking the label of a node x to be $\mu(V_{\mathcal{S}}(x))$ instead of $V_{\mathcal{S}}(x)$. Which interpretation is intended will be clear from the context.

Evidently, $(G_{\mathcal{S}}, w_0) \models \varphi$ iff $\langle T_{\mathcal{S}}, V_{\mathcal{S}} \rangle$ satisfies φ. Therefore, the model checking problem of $\mathcal{S}$ against φ can be reduced to check whether $\langle T_{\mathcal{S}}, V_{\mathcal{S}} \rangle$ belongs to the language of the *APT* (whose existence is guaranteed by Lemma 1) accepting the tree-models of φ. However, note that the branching degree of $T_{\mathcal{S}}$ is not uniform and, in particular, some nodes of $T_{\mathcal{S}}$ may not have successors. We solve this problem as follows. Let $k = max\{bd_{\mathcal{S}}(w) \mid w \in W\}$ (note that k is finite and can be trivially computed from the transition relation Δ of $\mathcal{S}$). We can encode the computation tree $\langle T_{\mathcal{S}}, V_{\mathcal{S}} \rangle$ as a $2^{AP \cup \{t\}} \cup \{\perp\}$-labelled complete k-ary tree (where $\perp$ and t are fresh proposition names not belonging to AP) in the following way: first, we add the proposition t to the label of all leaf nodes (corresponding to configurations without successors) of the tree $T_{\mathcal{S}}$; second, for each node $x \in T_{\mathcal{S}}$ with d children $1 \cdot x, \ldots, d \cdot x$ (note that $0 \leq d \leq k$), we add the children $(d+1) \cdot x, \ldots, k \cdot x$ and label these new nodes with $\perp$; finally, for each node x labelled by $\perp$ we add recursively k children labelled by $\perp$. Let $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle$ be the tree thus obtained. Since a node labelled by $\perp$ stands for a node that actually does not exist, we have to take this into account when we interpret CTL^* formulas over the tree $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle$. This means that we have to consider only the paths in this tree (called "legal" paths) that either never visit a node labelled by $\perp$ or contain a *terminal* node (i.e. a node labelled by t). Note that a path is *not* "legal" iff it satisfies the formula $\neg t\, \mathcal{U}\, \perp$. In order to achieve this, we define inductively a function $f : CTL^*$ formulas $\rightarrow CTL^*$ formulas such that $f(\varphi)$ restricts path quantification to only "legal" paths:

- $f(prop) = prop$ for any proposition $prop \in AP$;
- $f(\neg\varphi) = \neg f(\varphi)$;
- $f(\varphi_1 \wedge \varphi_2) = f(\varphi_1) \wedge f(\varphi_2)$;
- $f(E\varphi) = E((G\neg\perp) \wedge f(\varphi)) \;\vee\; E((F\ t) \wedge f(\varphi))$;
- $f(A\varphi) = A((\neg t\, \mathcal{U}\, \perp) \;\vee\; f(\varphi))$;
- $f(X\varphi) = X(f(\varphi) \wedge \neg\perp)$;
- $f(\varphi_1\, \mathcal{U}\, \varphi_2) = (f(\varphi_1) \wedge \neg\perp)\, \mathcal{U}\, (f(\varphi_2) \wedge \neg\perp)$.

Note that $|f(\varphi)| = O(|\varphi|)$. By definition of f, it follows that $\langle T_{\mathcal{S}}, V_{\mathcal{S}} \rangle$ satisfies φ (i.e., $(G_{\mathcal{S}}, w_0) \models \varphi$) *iff* $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle$ satisfies $f(\varphi)$.

Let $\mathcal{A}_{f(\varphi)} = \langle 2^{AP \cup \{t\}} \cup \{\perp\}, Q, q_0, \delta, F \rangle$, with $F = \{F_1, \ldots, F_m\}$, be the symmetric *APT* (whose existence is guaranteed by Lemma 1) accepting exactly the $2^{AP \cup \{t\}} \cup \{\perp\}$-labelled complete k-ary trees that satisfy $f(\varphi)$. We have to check whether $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle$ belongs to the language $\mathcal{L}(\mathcal{A}_{f(\varphi)})$. We reduce this problem to the emptiness of an *PD-APA* $\mathcal{P} = \langle \Gamma, (P \cup \{\perp\}) \times Q, (p_0, q_0), \alpha_0, \rho, F' \rangle$, which is defined as follows. The states of $\mathcal{P}$ consist either of pairs of states of $\mathcal{S}$ and states of $\mathcal{A}_{f(\varphi)}$, or pairs of the form $(\perp, q)$ where q is a state of $\mathcal{A}_{f(\varphi)}$. Intuitively, when the automaton $\mathcal{P}$ is in state $(p, q) \in P \times Q$ with stack content α, and (p, α) is a configuration associated with some node x of $\langle T_{\mathcal{S}}, V_{\mathcal{S}} \rangle$, then $\mathcal{P}$ simulates the behaviour of $\mathcal{A}$ starting from state q on the input tree given by the subtree of $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle$ rooted at node x. Moreover, in state $(\perp, q)$, $\mathcal{P}$ simulates the behaviour of $\mathcal{A}$ from state q on the input tree in which all nodes are labelled by $\perp$.

The parity acceptance condition F' is given by $\{(P \cup \{\perp\}) \times F_1, \ldots, (P \cup \{\perp\}) \times F_m\}$. Finally, the transition function ρ is defined as follows:

- for each $(p, q) \in P \times Q$ and $B \in \Gamma \cup \{\gamma_0\}$, $\rho((p, q), B)$ is defined as follows. Let $next_{\mathcal{S}}(p, B) = \{(p_1, \alpha_1), \ldots, (p_d, \alpha_d)\}$ (note that $0 \leq d \leq k$). If $d > 0$ (resp., $d = 0$), then $\rho((p, q), B)$ is obtained from formula $\delta(q, L(p, B))$ (resp., $\delta(q, L(p, B) \cup \{t\})$) by replacing each generator occurring in it of the form $\bigvee_{i=1}^{i=k}(i, q')$ with $\bigvee_{i=1}^{i=d}((p_i, q'), \alpha_i) \ \vee\ \bigvee_{i=d+1}^{i=k}((\bot, q'), \varepsilon)$, and each generator of the form $\bigwedge_{i=1}^{i=k}(i, q')$ with $\bigwedge_{i=1}^{i=d}((p_i, q'), \alpha_i) \ \wedge\ \bigwedge_{i=d+1}^{i=k}((\bot, q'), \varepsilon)$;
- for each $q \in Q$ and $B \in \Gamma \cup \{\gamma_0\}$, $\rho((\bot, q), B)$ is obtained from formula $\delta(q, \bot)$ by replacing each generator occurring in it of the form $\mathcal{C}_{i=1}^{i=k}(i, q')$, where $\mathcal{C} \in \{\bigvee, \bigwedge\}$, with $\mathcal{C}_{i=1}^{i=k}((\bot, q'), \varepsilon)$.

It is not hard to see that $\mathcal{P}$ has an accepting run iff $\langle [k]^*, \widetilde{V}_{\mathcal{S}} \rangle \in \mathcal{L}(\mathcal{A}_{f(\varphi)})$.

Note that the size $|\rho|$ of the transition function of $\mathcal{P}$ is bounded by $k \cdot |\delta| \cdot |\Delta|$. By Lemma 1, it follows that $\mathcal{P}$ has index $O(|\varphi|)$ and $|\rho|$ is bounded by $O(k \cdot 2^{|\varphi|} \cdot \Delta)$. Then, by Proposition 1 we obtain the main result of this section.

Theorem 1. *Given an* PDS $\mathcal{S} = \langle AP, \Gamma, P, \Delta, L \rangle$*, a configuration* w_0 *of* $\mathcal{S}$*, and a* CTL* *formula* φ*, the model checking problem of* $\mathcal{S}$ *with respect to* φ *is solvable in time* exponential *in* $k \cdot |\Delta| \cdot 2^{|\varphi|}$ *with* $k = max\{bd_{\mathcal{S}}(w) \mid w \in P \times \Gamma^* \cdot \gamma_0\}$.

5 Lower Bounds

In this section we give lower bounds for the PMC problem against CTL^* and for the program complexity of the PMC problem against CTL. The lower bound for CTL (resp., CTL^*) is shown by a reduction from the word problem for PSPACE–bounded (resp., EXPSPACE–bounded) alternating Turing Machines. Without loss of generality, we consider a model of alternation with a binary branching degree. Formally, an alternating Turing Machine (TM, for short) is a tuple $\mathcal{M} = \langle \Sigma, Q, Q_\forall, Q_\exists, q_0, \delta, F \rangle$, where Σ is the input alphabet, which contains the blank symbol #, Q is the finite set of states, which is partitioned into $Q = Q_\forall \cup Q_\exists$, $Q_\exists$ (resp., $Q_\forall$) is the set of existential (resp., universal) states, q_0 is the initial state, $F \subseteq Q$ is the set of accepting states, and the transition function δ is a mapping $\delta : Q \times \Sigma \rightarrow (Q \times \Sigma \times \{L, R\})^2$.

Configurations of $\mathcal{M}$ are words in $\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$. A configuration $\eta \cdot (q, \sigma) \cdot \eta'$ denotes that the tape content is $\eta\sigma\eta'$, the current state is q, and the reading head is at position $|\eta| + 1$. When $\mathcal{M}$ is in state q and reads an input $\sigma \in \Sigma$ in the current tape cell, then it nondeterministically chooses a triple (q', σ', dir) in $\delta(q, \sigma) = \langle (q_l, \sigma_l, dir_l), (q_r, \sigma_r, dir_r) \rangle$, and then moves to state q', writes σ' in the current tape cell, and its reading head moves one cell to the left or to the right, according to dir. For a configuration c, we denote by $succ_l(c)$ and $succ_r(c)$ the successors of c obtained choosing respectively the left and the right triple in $\langle (q_l, \sigma_l, dir_l), (q_r, \sigma_r, dir_r) \rangle$. The configuration c is *accepting* if the associated state q belongs to F. Given an input $x \in \Sigma^*$, a computation tree of $\mathcal{M}$ on x is a tree in which each node corresponds to a configuration. The root of the tree corresponds to the initial configuration associated with x.[3] A node that corresponds to a universal configuration (i.e., the associated state is in $Q_\forall$) has two

[3] We assume that initially $\mathcal{M}$'s reading head is scanning the first cell of the tape.

successors, corresponding to $succ_l(c)$ and $succ_r(c)$, while a node that corresponds to an existential configuration (i.e., the associated state is in $Q_\exists$) has a single successor, corresponding to either $succ_l(c)$ or $succ_r(c)$. The tree is *accepting* if all its paths (from the root) visit an accepting configuration. An input $x \in \Sigma^*$ is *accepted* by $\mathcal{M}$ if there exists an accepting computation tree of $\mathcal{M}$ on x.

If $\mathcal{M}$ is PSPACE–bounded (resp., EXPSPACE–bounded), then there is a constant $k \geq 1$ such that for each $x \in \Sigma^*$, the space needed by $\mathcal{M}$ on input x is bounded by $k \cdot |x|$ (resp., $2^{k \cdot |x|}$). It is well-known [CKS81] that EXPTIME (resp., 2EXPTIME) coincides with the class of all languages accepted by PSPACE–bounded (resp., EXPSPACE–bounded) alternating Turing Machines.

EXPTIME-hardness of the pushdown model checking problem against *CTL* was shown by Walukiewicz [Wal00] using a reduction from the word problem for PSPACE-bounded alternating Turing Machines. We use the basic ideas of the construction in [Wal00] in order to prove that the program complexity of the problem (i.e., assuming the *CTL* formula is fixed) is still EXPTIME-hard.

Theorem 2. *The program complexity of the PMC problem for* CTL *is* EXPTIME-*hard.*

Proof. We show that there is a *CTL* formula φ such that given a PSPACE–bounded alternating Turing Machine $\mathcal{M} = \langle \Sigma, Q, Q_\forall, Q_\exists, q_0, \delta, F \rangle$ and an input x, it is possible to define an *PDS* $\mathcal{S}$ and a configuration w of $\mathcal{S}$, whose sizes are *polynomial* in $n = k \cdot |x|$ and in $|\mathcal{M}|$,[4] such that $\mathcal{M}$ accepts x *iff* $(G_\mathcal{S}, w) \models \varphi$.

Note that any reachable configuration of $\mathcal{M}$ over x can be seen as a word in $\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$ of length exactly n. If $x = \sigma_1 \ldots \sigma_r$ (where $r = |x|$), then the initial configuration is given by $(q_0, \sigma_1)\sigma_2 \ldots \sigma_r \underbrace{\#\# \ldots \#}_{n-r}$.

$\mathcal{S}$ guesses accepting computation trees of $\mathcal{M}$ starting from TM configurations of length n. The internal nodes of these trees are non-accepting configurations and the leaves are accepting configurations. The trees are traversed as follows. If the current non-accepting configuration c is universal, then $\mathcal{S}$, first, will examine the subtrees associated with the left successor of c, and successively the subtrees associated with the right successor. If instead c is existential, then $\mathcal{S}$ will guess one of the two successors of c and, consequently, it will examine only the subtrees associated with this successor. In order to guess an accepting tree (if any) from a given configuration, $\mathcal{S}$ keeps track on the stack of the path from the root to the actual TM configuration by pushing the new guessed configurations and popping when backtracking along the accepting subtree guessed so far. Therefore, $\mathcal{S}$ accepts by empty stack. The stack alphabet of $\mathcal{S}$ is given by $\Sigma \cup (Q \times \Sigma) \cup \{\exists_l, \exists_r, \forall_l, \forall_r\}$ where $\exists_l$ and $\exists_r$ (resp., $\forall_l$ and $\forall_r$) are used to delimitate the left and right successors of an existential (resp., universal) configuration. The behaviour of $\mathcal{S}$ can be subdivided in three steps.

1. *Generation of a TM configuration* (*operative phase*) - $\mathcal{S}$ generates nondeterministically by push transitions a TM configuration c followed by a symbol

[4] Where $k \geq 1$ is a constant such that for each input $y \in \Sigma^*$, the space needed by $\mathcal{M}$ on input y is bounded by $k \cdot |y|$.

in $\{\forall_l, \exists_l, \exists_r\}$ on the stack, with the constraint that $\forall_l$ is chosen iff c is a universal configuration (i.e., the TM state q associated with c belongs to $Q_\forall$). In this phase, a (control) state of $\mathcal{S}$ has the form $(gen, q, i, flag)$, where $q \in Q$ keeps track of the TM state associated with c, gen is a label identifying the current operation of $\mathcal{S}$, $i \in \{0, \ldots, n+1\}$ is used to ensure that c has exactly length n, and $flag \in \{0, 1\}$ is used to ensure that $c \in \Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$.

When $\mathcal{S}$ finishes to generate a TM configuration c followed by a symbol $m \in \{\forall_l, \exists_l, \exists_r\}$, i.e. $\mathcal{S}$ is in a state of the form $(gen, q, n+1, 1)$, then it chooses nondeterministically between two possible options. Choosing the first option, $\mathcal{S}$ goes to state *cont*, pops m from the stack, and performs Step 3 (see below). Choosing the second option, the behaviour of $\mathcal{S}$ depends on whether c is accepting. If c is *not* accepting (i.e., $q \notin F$), then $\mathcal{S}$ guesses a successor of c going to a state of the form $(gen, q', 0, 0)$ for some $q' \in Q$ without changing the stack content. Therefore, Step 1 is newly performed. If instead c is accepting (i.e., $q \in F$), then $\mathcal{S}$ goes to state *rem*, pops m from the stack, and performs Step 2 (see below).

2. *Removing a TM configuration* (*operative phase*) - When $\mathcal{S}$ is in state *rem*, it removes deterministically by pop transitions the TM configuration c on the top of the stack (if any). After having removed c, if the symbol on the top of the stack, say B, belongs to $\{\forall_r, \exists_l, \exists_r\}$ (this means intuitively that $\mathcal{S}$ has already generated a "pseudo" accepting computation tree for the TM configuration currently on the top of the stack), then $\mathcal{S}$ pops B from the stack and goes to state *rem* (i.e., Step 2.2 is newly performed). If instead $B = \forall_l$, then $\mathcal{S}$ goes to a state of the form $(gen, q', 0, 0)$ for some $q' \in Q$ and replaces $\forall_l$ with the symbol $\forall_r$ on the top of the stack. Therefore, Step 1 is newly performed. Finally, if $B = \gamma_0$ (i.e., the stack is empty), then $\mathcal{S}$ goes to state fin and terminates its computation.
3. *Checking δ-consistency* (*control phase*)- When $\mathcal{S}$ is in state *cont*, it checks that one of the following holds:
   - the stack contains exactly one TM configuration.
   - the stack content has the form $c' \cdot m \cdot c \cdot \alpha$ where c and c' are TM configurations and $m \in \{\exists_l, \exists_r, \forall_l, \forall_r\}$.

   In the first case, $\mathcal{S}$ signals success by generating (by its finite control) the symbol *good*. In the second case, $\mathcal{S}$ signals success if and only if c' is a TM successor of c in accordance with m, i.e.: $c' = succ_s(c)$ where $s = l$ *iff* $m \in \{\exists_l, \forall_l\}$. In order to understand how this can be done by using a number of states polynomial in n and $|\mathcal{M}|$, let $c = a_1 \ldots a_n$. For each $1 \leq i \leq n$, the value a'_i of the i-th cell of $succ_l(c)$ (resp., $succ_r(c)$) is completely determined by the values a_{i-1}, a_i and a_{i+1} (taking a_{n+1} for $i = n$ and a_0 for $i = 1$ to be some special symbol, say "$-$"). As in [KTMV00], we denote by $next_l(a_{i-1}, a_i, a_{i+1})$ (resp., $next_r(a_{i-1}, a_i, a_{i+1})$) our expectation for a'_i (these functions can be trivially obtained from the transition function of $\mathcal{M}$). Then, in state *cont*, $\mathcal{S}$ chooses nondeterministically between n states, $cont_1, \ldots, cont_n$ without changing the stack content. For each $1 \leq i \leq n$, if $\mathcal{S}$ is in state $cont_i$, then first, it *deterministically* removes $c' \cdot m$ from the stack, keeping track by its finite control of m and the i-th symbol a'_i of

c'. Successively, $\mathcal{S}$ *deterministically* removes c from the stack, keeping also track of the symbols a_{i-1}, a_i, and a_{i+1}. Finally, $\mathcal{S}$ checks whether $a'_i = next_s(a_{i-1}, a_i, a_{i+1})$ with $s = l$ *iff* $m \in \{\exists_l, \forall_l\}$. If this condition is satisfied (and only in this case), then $\mathcal{S}$ generates the symbol *good* and terminates the computation.

Formally, $\mathcal{S} = \langle AP, \Gamma, P, \Delta, L\rangle$ is defined as follows:

- $AP = \{op, cont, good, fin\}$ and $\Gamma = \Sigma \cup (Q \times \Sigma) \cup \{\forall_l, \forall_r, \exists_l, \exists_r\}$;
- $P = \{good, fin, rem\} \cup P_G \cup P_\delta$ where $P_G = \{(gen, q, i, flag) \mid q \in Q,\ 0 \leq i \leq n+1,\ flag \in \{0,1\},\ flag = 0$ if $i = 0$ and $flag = 1$ if $i = n, n+1\}$ is the set of (control) states used in Step 1, and P_δ, which is used in Step 3, is given by
$$\{cont, cont_1, \ldots, cont_n\} \cup \{(cont_i, j, a) \mid 1 \leq i, j \leq n \text{ and } a \in \Sigma \cup (Q \times \Sigma)\}$$
$$\cup\{(cont_i, j, a, m, a_1, a_2, a_3) \mid 1 \leq i \leq n,\ 0 \leq j \leq n,\ m \in \{\forall_l, \forall_r, \exists_l, \exists_r\},$$
$$a, a_1, a_2, a_3 \in \Sigma \cup (Q \times \Sigma) \cup \{-\}, \text{ and } a \neq -\}$$
- $((p, B), (p', \beta)) \in \Delta$ iff one of the following holds:
  - *Step 1* (*generation of a TM configuration*) - If $p \in P_G$, then:
    - if $p = (gen, q, i, flag)$ and $i < n$, then $\beta = B'B$ with $B' \in \Sigma \cup (\{q\} \times \Sigma)$ and $p' = (gen, q, i+1, flag')$. Moreover, if $flag = 1$, then $B' \in \Sigma$ and $flag' = 1$; otherwise, $flag' = 0$ iff $B' \in \Sigma$.
    - if $p = (gen, q, n, 1)$, then $p' = (gen, q, n+1, 1)$ and $\beta = B'B$ with $B' = \forall_l$ if $q \in Q_\forall$, and $B' \in \{\exists_l, \exists_r\}$ otherwise.
    - if $p = (gen, q, n+1, 1)$, then or (1) $\beta = \varepsilon$ and $p' = cont$, or (2) $q \notin F$, $\beta = B \in \Gamma$, and $p' = (gen, q', 0, 0)$ for some $q' \in Q$, or (3) $q \in F$, $\beta = \varepsilon$, and $p' = rem$.
  - *Step 2* (*Removing a TM configuration*). If $p = rem$, then:
    - if $B \in \Sigma \cup (Q \times \Sigma) \cup \{\forall_r, \exists_l, \exists_r\}$, then $\beta = \varepsilon$ and $p' = rem$;
    - if $B = \forall_l$, then $\beta = \forall_r$ and $p' = (gen, q', 0, 0)$ for some $q' \in Q$;
    - if $B = \gamma_0$, then $\beta = \varepsilon$, and $p' = fin$.
  - *Step 3* (*Checking δ-consistency*). If $p \in P_\delta$, then:
    - if $p = cont$, then $\beta = B$ and $p' = cont_i$ for some $1 \leq i \leq n$.
    - if $p = cont_i$, then $B \in \Sigma \cup (Q \times \Sigma)$, $\beta = \varepsilon$, and $p' = (cont_i, 1, B)$;
    - if $p = (cont_i, j, a)$ and $j < n$, then $B \in \Sigma \cup (Q \times \Sigma)$, $\beta = \varepsilon$, and $p' = (cont_i, j+1, a')$ where $a' = B$ if $j = i-1$, and $a' = a$ otherwise;
    - if $p = (cont_i, n, a)$, then *either* $B = \gamma_0$, $\beta = \varepsilon$, and $p' = good$, *or* $B \in \{\exists_l, \forall_l, \exists_r, \forall_r\}$, $\beta = \varepsilon$, and $p' = (cont_i, 0, a, B, -, -, -)$;
    - if $p = (cont_i, j, a, m, a_1, a_2, a_3)$ and $j < n$, then $B \in \Sigma \cup (Q \times \Sigma)$, $\beta = \varepsilon$, and $p' = (cont_i, j+1, a, m, a'_1, a'_2, a'_3)$ where for each $1 \leq h \leq 3$, $a'_h = B$ if $j = i + h - 3$, and $a'_h = a_h$ otherwise;
    - if $p = (cont_i, n, a, m, a_1, a_2, a_3)$, then $a = next_s(a_1, a_2, a_3)$ where $s = l$ if and only if $m \in \{\exists_l, \forall_l\}$. Moreover, $\beta = \varepsilon$ and $p' = good$.
- For all $B \in \Gamma \cup \{\gamma_0\}$, $L(good, B) = \{good\}$, $L(fin, B) = \{fin\}$, $L(rem, B) = op$, for all $p \in P_G$, $L(p, B) = \{op\}$, and for all $p \in P_\delta$, $L(p, B) = \{cont\}$.

Let $G_{\mathcal{S}} = \langle W, R, \mu\rangle$. The correctness of the construction is stated by the following claim:

Claim. Given a TM configuration c with TM state q, there is an accepting computation tree of $\mathcal{M}$ over c *iff* there is a path of $G_{\mathcal{S}}$ of the form $\pi = w_0 w_1 \ldots w_n$ such that $w_0 = ((gen, q, n, 1), c \cdot \gamma_0)$, $\mu(w_n) = fin$, and for each $0 \leq i \leq n-1$, $\mu(w_i) = op$ and if w_i has a successor w_i' such that $\mu(w_i') = cont$, then each path from w_i' visits a state of the form $(good, \beta)$.

The condition in the claim above can be encoded by the following *CTL* formula

$$\varphi := E\Big(\big[op \wedge AX(cont \rightarrow AF good)\big]\ \mathcal{U}\ fin\Big) \tag{1}$$

Let c_0 be the initial TM configuration (associated with the input x). Then, by Claim 1 it follows that $\mathcal{M}$ accepts x *iff* $(G_{\mathcal{S}}, w) \models \varphi$ where $w = ((gen, q_0, n, 1), c_0 \cdot \gamma_0)$. Since φ is independent from $\mathcal{M}$ and n, and the sizes of $|\mathcal{S}|$ and w are polynomial in n and $|\mathcal{M}|$, the assertion holds. □

Theorem 3. *Pushdown model checking against* CTL* *is* 2EXPTIME-*hard.*

Proof. Let $\mathcal{M} = \langle \Sigma, Q, Q_\forall, Q_\exists, q_0, \delta, F \rangle$ be an EXPSPACE–bounded alternating Turing Machine, and let k be a constant such that for each $x \in \Sigma^*$, the space needed by $\mathcal{M}$ on input x is bounded by $2^{k \cdot |x|}$. Given an input $x \in \Sigma^*$, we define an *PDS* $\mathcal{S}$, a configuration $w_0 = (p_0, \gamma_0)$ of $\mathcal{S}$, and a CTL^* formula φ, whose sizes are *polynomial* in $n = k \cdot |x|$ and in $|\mathcal{M}|$, such that $\mathcal{M}$ accepts x *iff* $(G_{\mathcal{S}}, w_0) \models \varphi$. Some ideas in the proposed reduction are taken from [KTMV00], where there are given lower bounds for the satisfiability of extensions of *CTL* and CTL^*.

Note that any reachable configuration of $\mathcal{M}$ over x can be seen as a word in $\Sigma^* \cdot (Q \times \Sigma) \cdot \Sigma^*$ of length exactly 2^n. If $x = \sigma_1 \ldots \sigma_r$ (where $r = |x|$), then the initial configuration is given by $(q_0, \sigma_1)\sigma_2 \ldots \sigma_r \underbrace{\#\# \ldots \#}_{2^n - r}$.

Each cell of a TM configuration is coded using a block of n symbols of the stack alphabet of $\mathcal{S}$. The whole block is used to encode both the content of the cell and the location (the number of cell) on the TM tape (note that the number of cell is in the range $[0, 2^n - 1]$ and can be encoded using n bits). The stack alphabet is given by $\{\forall_l, \forall_r, \exists_l, \exists_r\} \bigcup (\Sigma \cup (Q \times \Sigma)) \times 2^{\{b,e,f,cn,l\}}$ where b is used to mark the first element of a TM block, e (resp., f) to mark the first (resp., the last) block of a TM configuration, cn to encode the number of cell, and l to mark a left TM successor.

The behaviour of $\mathcal{S}$ is similar to that of the pushdown system defined in the proof of Theorem 2. The main differences can be summarized as follows:

- *Generation of a TM configuration* (**Step 1**) When $\mathcal{S}$ generates nondeterministically a TM configuration c on the stack, it ensures that each block of c has length n and the symbols b, f, and e are used properly. Moreover, if c is generated as a successor of an other TM configuration, i.e. the stack content before generating c has the form $m \cdot \alpha$ with $m \in \{\exists_l, \exists_r, \forall_l, \forall_r\}$, then $\mathcal{S}$ ensures that the label l is used properly, i.e. any element of c is marked by l iff $m \in \{\exists_l, \forall_l\}$. However, $\mathcal{S}$ does *not* ensure the the cell numbers of c are encoded properly (indeed, this would require a number of control states exponential in n).

- *Generation of the initial TM configuration* - Starting from the global state $w_0 = (p_0, \gamma_0)$, $\mathcal{S}$, first, generates the encoding of the initial TM configuration c_0 (associated with the input x) on the stack. Note that $\mathcal{S}$ ensures that c_0 has the form $(q_0, \sigma_1)\sigma_2 \ldots \sigma_r \# \# \ldots$. However, $\mathcal{S}$ does not ensure that the number of blanks to the right of σ_r is exactly $2^n - r$.
- *Checking δ-consistency* - As for the pushdown system defined in the proof of Theorem 2, after having generated a TM configuration on the stack, $\mathcal{S}$ can choose nondeterministically to go to the (control) state *cont*. When $\mathcal{S}$ is in state *cont*, it chooses nondeterministically between two options $cont_1$ and $cont_2$ (without changing the stack content). Assume that the stack content has the form $c \cdot \alpha$ where c is a "pseudo" TM configuration generated in Step 1, and either α is empty or it has the form $m \cdot c' \cdot \alpha'$ where $m \in \{\exists_l, \exists_r, \forall_l, \forall_r\}$ and c' is a "pseudo" TM configuration. Then, choosing option $cont_1$, $\mathcal{S}$ removes deterministically (by pop transitions) c from the stack and terminates its computation. The computation tree $\langle T, V \rangle$ of $G_{\mathcal{S}}$ rooted at the global state associated with $cont_1$ reduces to a finite path π (corresponding to the configuration c). We use a CTL^* formula φ_1 on this tree $\langle T, V \rangle$ in order to require that the cell numbers of c are encoded correctly (this also implies that the number of blocks of c is exactly 2^n). For each node $u \in \pi$, let $cn(u)$ be the truth value (1 for *true* and 0 for *false*) of the proposition cn in u. Let us consider two consecutive TM blocks $u_1 \ldots u_n u'_1 \ldots u'_n$ along π, and let k (resp., k') be the number of cell of the first block (resp., the second block), i.e., the integer whose binary code is given by $cn(u_1) \ldots cn(u_n)$ (resp., $cn(u'_1) \ldots cn(u'_n)$). We have to require that $k' = (k+1)\ mod\ 2^n$, and $k = 0$ (resp., $k' = 2^n - 1$) if $u_1 \ldots u_n$ corresponds to the first block of c, i.e. u_1 is labelled by proposition e (resp., $u'_1 \ldots u'_n$ corresponds to the last block of c, i.e. u'_1 is labelled by proposition f). Therefore, φ_1 is defined as follows:

$$
\begin{aligned}
AG\ \Big(\ & \big((b \wedge e) \rightarrow \textstyle\bigwedge_{j=0}^{n-1} (AX)^j \neg cn\big) \ \bigwedge \ \big((b \wedge f) \rightarrow \textstyle\bigwedge_{j=0}^{n-1} (AX)^j\ cn\big) \ \bigwedge \\
& \big[(b \wedge \neg f) \longrightarrow \textstyle\bigvee_{j=0}^{n-1} [(AX)^j (\neg cn \wedge (AX)^n cn) \wedge \\
& \textstyle\bigwedge_{i>j} (AX)^i (cn \wedge (AX)^n \neg cn) \wedge \textstyle\bigwedge_{i<j} (AX)^i (cn \leftrightarrow (AX)^n cn)]\big] \ \Big)
\end{aligned}
$$

Choosing the second option $cont_2$, $\mathcal{S}$, first, removes deterministically c from the stack with the additional ability to generate (by its finite control) the symbol $check_1$. Successively, assuming that α has the form $m \cdot c' \cdot \alpha'$, $\mathcal{S}$ removes $m \cdot c'$ from the stack (by pop transitions) and simultaneously generates (by its finite control) at most at one block of c' the symbol $check_2$. After this operation, $\mathcal{S}$ terminates its computation. Let $\langle T, V \rangle$ be the computation tree of $G_{\mathcal{S}}$ rooted at the global state associated with $cont_2$. If α is empty, then by construction, T reduces to a finite path labelled by proposition $check_1$ and corresponding to configuration c. If instead α has the form $m \cdot c' \cdot \alpha'$, then each path (from the root) of T consists of a sequence of nodes corresponding to c labelled by $check_1$ followed by a sequence of nodes corresponding to c' with at most one block labelled by $check_2$. This allows us to define a CTL^* formula φ_2, asserted on the tree $\langle T, V \rangle$, (whose size is polynomial in n and $|\mathcal{M}|$) in order to require that in the case α is not empty (i.e., α has the form

$m \cdot c' \cdot \alpha'$), c is a TM successor of c' in accordance with m, i.e. $c = succ_s(c')$ where $s = l$ iff $m \in \{\exists_l, \forall_l\}$ (note that by Step 1, $m \in \{\exists_l, \forall_l\}$ iff c is marked by symbol l). Formula φ_2 is defined as follows:

$$AG(\neg check_2) \ \vee \ AG((check_1 \ \wedge \ b) \rightarrow E(\theta_1 \ \wedge \ \theta_2))$$

where the path formulas θ_1 and θ_2 are defined below. Note that the subformula $AG(\neg check_2)$ manages the case in which α is empty. In the other case, we require that for each node $u \in T$ labelled by $check_1$ and b, i.e. associated with the first element of a block bl of c, there is a path π from u satisfying the following two properties:

1. π visits a node labelled by $check_2$ and b, i.e. associated with the first element of a block bl' of c', such that bl and bl' have the same number of cell. This requirement is specified by the path formula θ_1:

   $$\theta_1 := \psi_1 \ \wedge \ X(\psi_2 \ \wedge \ X(\psi_3 \ \wedge \ldots X(\psi_n)\ldots))$$

   where for each $1 \leq j \leq n$, ψ_j is defined as follows

   $$\big(cn \ \rightarrow \ F(check_2 \wedge b \wedge X^{j-1}\, cn)\big) \wedge \big(\neg cn \ \rightarrow \ F(check_2 \wedge b \wedge X^{j-1} \neg\, cn)\big)$$

2. Let $\Sigma' := \Sigma \cup (Q \times \Sigma)$ and let us denote by $\sigma(\widehat{bl})$ the Σ'-value of a TM block $\widehat{bl}$. By construction and Property 1 above, there is exactly one node of π that is labelled by $check_2$ and b. Moreover, by Property 1 this node is associated with a TM block bl' of c' having the same number of cell of bl. Therefore, we have to require that $\sigma(bl) = next_s(\sigma(bl_{prec}), \sigma(bl'), \sigma(bl_{succ}))$ where bl_{prec} and bl_{succ} represent the blocks soon before and soon after bl' along π, and $s = l$ iff the TM configuration c is a left TM successor (i.e. all nodes of bl are labelled by proposition l). This requirement is expressed by the path formula θ_2. We distinguish three cases depending on whether bl corresponds to the first block, to the last block or to a non-extremal block of the associated TM configuration c. For simplicity, we consider only the case in which bl is a non-extremal block. The other cases can be handled similarly.

$$\begin{aligned}\theta_2 := (\neg f \ \wedge \ \neg e) \longrightarrow &\\ \textstyle\bigvee_{\sigma_1,\sigma_2,\sigma_3 \in \Sigma} \big(F(\sigma_1 \ \wedge (X)^n(\sigma_2 \ \wedge b \ \wedge \ check_2 \ \wedge \ (X)^n\sigma_3)) \ \textstyle\bigwedge \\ (l \ \rightarrow next_l(\sigma_1,\sigma_2,\sigma_3)) \ \textstyle\bigwedge \ (\neg l \ \rightarrow next_r(\sigma_1,\sigma_2,\sigma_3))\big)\end{aligned}$$

Finally, formula φ is obtained from formula (1) in the proof of Theorem 2 by replacing the subformula $AX(cont \rightarrow AF good)$ in (1) with the formula $AX\big[cont \rightarrow \big(EX(cont_1 \wedge \varphi_1) \wedge EX(cont_2 \wedge \varphi_2)\big)\big]$. □

Now, we can prove the main result of this paper.

Theorem 4.
(1) *The program complexity of the PMC problem for* CTL *is* EXPTIME-*complete.*
(2) *The PMC problem for* CTL* *is* 2EXPTIME-*complete. The program complexity of the problem is* EXPTIME-*complete.*

Proof. Claims 1 follows from Theorem 2 and the fact that model-checking pushdown systems against *CTL* is known to be EXPTIME-complete [Wal00], while Claim 2 directly follows from Theorems 1 and 3, and Claim 1. □

References

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal Logic of Nested Calls and Returns. In *TACAS'04*, pages 467–481, 2004.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR'97*, LNCS 1243, pages 135–150. Springer-Verlag, 1997.

[Cau96] D. Caucal. On infinite transition graphs having a decidable monadic theory. In *Proc. the 23th International Colloquium on Automata, Languages and Programming (ICALP'96)*, LNCS 1099, pages 194–205. Springer-Verlag, 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and verification of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV'00*, LNCS 1855, pages 232–247. Springer-Verlag, 2000.

[EJ91] E.A. Emerson and C.S. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS'91*, pages 368–377, 1991.

[EKS03] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[KPV02] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *LPAR'02*, LNCS 2514, pages 262–277. Springer-Verlag, 2002.

[KTMV00] O. Kupferman, P.S. Thiagarajan, P. Madhusudan, and M.Y. Vardi. Open systems in reactive environments: Control and Synthesis. In *CONCUR'00*, LNCS 1877, pages 92–107. Springer-Verlag, 2000.

[KVW00] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.

[MS85] D.E. Muller and P.E. Shupp. The theory of ends, pushdown automata, and second-order logic. *Theoretical Computer Science*, 37:51–75, 1985.

[MS87] D.E. Muller and P.E. Shupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[PV04] N. Piterman and M.Y. Vardi. Global model-checking of infinite-state systems. In *CAV'04*, LNCS 3114, pages 387–400. Springer-Verlag, 2004.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In *POPL'88*, pages 250–259. ACM Press, 1988.

[Var98] M.Y. Vardi. Reasoning about the past with two-way automata. In *ICALP'98*, LNCS 1443, pages 628–641. Springer-Verlag, 1998.

[Wal96] I. Walukiewicz. Pushdown processes: Games and Model Checking. In *CAV'96*, LNCS 1102, pages 62–74. Springer-Verlag, 1996.

[Wal00] I. Walukiewicz. Model checking CTL properties of pushdown systems. In *FSTTCS'00*, LNCS 1974, pages 127–138. Springer-Verlag, 2000.

A Compositional Logic for Control Flow

Gang Tan[1] and Andrew W. Appel[2]

[1] Computer Science Department, Boston College
gtan@cs.bc.edu
[2] Computer Science Department, Princeton University
appel@cs.princeton.edu

Abstract. We present a program logic, $\mathcal{L}_c$, which modularly reasons about unstructured control flow in machine-language programs. Unlike previous program logics, the basic reasoning units in $\mathcal{L}_c$ are multiple-entry and multiple-exit program fragments. $\mathcal{L}_c$ provides fine-grained composition rules to compose program fragments. It is not only useful for reasoning about unstructured control flow in machine languages, but also useful for deriving rules for common control-flow structures such as while-loops, repeat-until-loops, and many others. We also present a semantics for $\mathcal{L}_c$ and prove that the logic is both sound and complete with respect to the semantics. As an application, $\mathcal{L}_c$ and its semantics have been implemented on top of the SPARC machine language, and are embedded in the Foundational Proof-Carrying Code project to produce memory-safety proofs for machine-language programs.

1 Introduction

Hoare Logic [1] has long been used to verify properties of programs written in high-level programming languages. In Hoare Logic, a triple $\{p\}s\{q\}$ describes the relationship between exactly two states—the normal entry and exit states—associated with a program execution. That is, if the state before execution of s satisfies the assertion p, then the state after execution satisfies q. For a high-level programming language with structured control flow, a program logic based on Hoare triples works fine.

However, programs in high-level languages are compiled into machine code to execute. Since it is hard to prove that a compiler with complex optimizations produces correct machine code from verified high-level-language programs, substantial research effort $[2, 3, 4]$ during recent years has been devoted to verifying properties directly at the machine-language level.

Machine-language programs contain goto statements with unrestricted destinations. Therefore, a program fragment or a collection of statements possibly contains multiple exits and multiple entries to which goto statements might jump. In Hoare Logic, since a triple $\{p\}s\{q\}$ is tailored to describe the relationship between the normal entry and the normal exit states, it is not surprising that trouble arises in considering program fragments with more than one entry/exit.

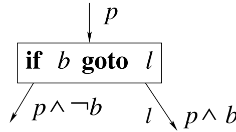
To address the problem of reasoning about control flow in machine-language programs, this paper makes two main contributions:

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 80–94, 2006.
© Springer-Verlag Berlin Heidelberg 2006

– We propose a program logic, $\mathcal{L}_c$, which modularly reasons about machine-language program fragments. Its basic reasoning units are multiple-entry and multiple-exit program fragments. The logic composes program fragments in a set of fine-grained composition rules. As a result, $\mathcal{L}_c$ is more modular than previous program logics for control flow.
– We also develop for $\mathcal{L}_c$ a semantics. We will show that a naive semantics does not work. We need to use a semantics based on approximations of counting computation steps. Based on this semantics, soundness and (relative) completeness of $\mathcal{L}_c$ are proved.

Before a full technical development, we present an overview of $\mathcal{L}_c$ and its related work.

Overview of $\mathcal{L}_c$. Two design features give $\mathcal{L}_c$ its modularity: its judgment (form of specification) and its composition rules. The judgment in $\mathcal{L}_c$ is directly on multiple-entry and multiple-exit program fragments. For example, $\mathcal{L}_c$ treats a conditional-branch statement "**if** b **goto** l" as a one-entry and two-exit fragment. $\mathcal{L}_c$ then provides for "**if** b **goto** l" a rule, which associates the entries and exits with appropriate invariants, depicted as follows:



The above graph associates the invariant p with the entry, and associates $p \wedge \neg b$ and $p \wedge b$ with two exits, respectively. As a note to our convention, we put invariants on the right of edges; we put labels, when they exist, on the left.

$\mathcal{L}_c$ also provides a set of inference rules to compose judgments on program fragments. These inference rules reason about control flow in smaller steps than Hoare Logic. For example, to reason about while loops, Hoare Logic provides a while rule:

$$\frac{\{p \wedge b\}s\{p\}}{\{p\}\textbf{while } b \textbf{ do } s\{p \wedge \neg b\}} \text{ while} \qquad \begin{array}{l} l: \ \textbf{if } \neg b \textbf{ goto } l'; \\ l_1 : s; \\ l_2 : \textbf{goto } l \\ l' : \end{array} \tag{1}$$

A while loop, however, is a high-level language construct. When mapped to machine code, it is implemented by a sequence of more primitive statements. One implementation is shown on the right of the previous figure. Since the implementation contains unstructured control flow, Hoare logic cannot reason about it. In contrast, our logic can treat each statement in the implementation as a multiple-entry and multiple-exit fragment. Using its composition rules, the logic can combine fragments and eliminate intermediate entries and exits. In the end, from its composition rules, the logic can derive the Hoare-logic rule for while loops. Furthermore, it can derive the rules for sequential composition,

repeat-until loops, if-then-else statements, and many other structured control-flow constructs. Therefore, our logic can recover structured control flow, when present, in machine-language programs.

Related work on program logics for goto statements. Many researchers have also realized the difficulty of verifying properties of programs with goto statements in Hoare Logic [5, 6, 7, 8, 9]. Some of them have proposed improvements over Hoare Logic. Almost all of these works are at the level of high-level languages. They treat while loops as a separate syntactic construct and have a rule for it. In comparison, $\mathcal{L}_c$ derives rules for control-flow structures.

These previous works also differ from $\mathcal{L}_c$ in terms of the form of specification. The work by de Bruin [8] is a typical example. In his system, the judgment for a statement s is:

$$\langle L_1 : p_1, \ldots, L_n : p_n | \{p\} s \{q\} \rangle, \tag{2}$$

where $L_1, \ldots, L_n$ are labels in a program P; the assertion p_i is the invariant associated with the label L_i; the statement s is a part of the program P. Judgment (2) judges a triple $\{p\}s\{q\}$, but under all label invariants in a program. By explicitly supplying invariants for labels in the judgment, de Bruin's system can handle goto statements, and its rule for goto statements is $\langle L_1 : p_1, \ldots, L_n : p_n | \{p_i\} \textbf{goto } L_i \{\mathit{false}\} \rangle$.

Judgment (2) is sufficient for verifying properties of programs with goto statements. Typed Assembly Language (TAL [3]) by Morrisett et al. uses a similar judgment to verify type safety of assembly-language programs. However, judgment (2) assumes the availability of global information, because it judges a statement s under all label invariants of a program— $L_1 : p_1, \ldots, L_n : p_n$. Consequently, it is impossible for de Bruin's system or TAL to compose fragments with different sets of global label invariants. We believe that a better form of specification should judge s under only those label invariants associated with exits in s. This new form of specification makes fewer assumptions (fewer label invariants) about the rest of the program and is more modular.

Floyd's work [10] on program verification associates a predicate for each arc in the flowchart representation of a program. The program is correct if each statement in the program has been verified correct with respect to the predicates associates with the entry and exit arcs of the statement. In Floyd's system, however, the composition of statements is based on flowcharts and is informal, and it has no principles for eliminating intermediate arcs. Our $\mathcal{L}_c$ provides formal rules for composing statements. When verifying properties of goto statements and labels, Floyd's system also assumes the availability of the complete program.

Cardelli proposed a linking logic [11] to formalize program linking. Glew and Morrisett [12] defined a modular assembly language to perform type-safe linking. Our logic is related to these works because exit labels can be thought as imported labels in a module, and entry labels as exported labels. In some sense, we apply the idea of modular linking to verification of machine code. But since we are more concerned with program verification, we also provide a semantics for our logic, and prove it is both sound and complete.

Recent works by Benton [13], Ni and Shao [14], and Saabas and Usstalu [15] define compositional program logics for low-level machines; their systems also reason modularly about program fragments and linking. To deal with procedure calls and returns, Benton uses Hoare-style pre- and postconditions. Since our compiler uses continuation-passing style, so can our calculus; therefore our labels need only preconditons.

The rest of this paper is organized as follows. Section 2 presents the logic $\mathcal{L}_c$ on a simple imperative language that has unstructured control flow. In Section 3, we develop a semantics for the logic. The soundness and completeness theorems are then presented. In Section 4, we briefly discuss the implementation and the role of $\mathcal{L}_c$ in the Foundational Proof-Carrying Code project [4]. In Section 5, we conclude and discuss future work. A more detailed treatment of the logic, its semantics, and its applications can be found in the first author's PhD thesis [16].

2 Program Logic $\mathcal{L}_c$

We present $\mathcal{L}_c$ on a simple imperative language. Figure 1 presents the syntax of the language. Most of the syntax is self-explanatory, and we only stress a few points. First, since the particular set of primitive operators and relations does not affect the presentation, the language assumes a class of operator symbols, *OPSym*, and a class of relation symbols, *RSym*. For concreteness, *OPSym* could be $\{+, \times, 0, 1\}$ and *RSym* could be $\{=, <\}$. Second, boolean operators do not include standard constructors such as **false**, $\wedge$ and $\Rightarrow$; they can be defined by **true**, $\vee$ and $\neg$.

The language in Fig. 1 is tailored to imitate a machine language. The destination of a goto statement is unrestricted and may be a label in the middle of a loop. Furthermore, the language does not have control structures such as **if** b **then** s, and **while** b **do** s. These control structures are implemented by a sequence of primitive statements.

To simplify the presentation, the language in Fig. 1 differs from machine languages in several aspects. It uses abstract labels while machine languages use concrete addresses. This difference does not affect the results of $\mathcal{L}_c$. The language also lacks indirect jumps (jump through a variable), pc-relative jumps, and procedure calls. We will discuss in Section 4 how we deal with these features.

operator symbols	*OPSym*	op	
relation symbols	*RSym*	re	
variables	*Var*	x, y, z	
labels	*Label*	l	
primitive statements	*PrimStmt*	t	$::= x := e \mid \textbf{goto}\ l \mid \textbf{if}\ b\ \textbf{goto}\ l$
statements	*Stmt*	s	$::= t \mid l : s \mid (s_1; s_2)$
expressions	*Exp*	e	$::= x \mid op(e_1, \ldots, e_{ar(op)})$
boolean expressions	*BExp*	b	$::= \textbf{true} \mid b_1 \vee b_2 \mid \neg b \mid re(e_1, \ldots, e_{ar(re)})$

Fig. 1. Language syntax, where $ar(op)$ is the arity of the symbol op

2.1 Syntax and Rules of $\mathcal{L}_c$

The syntax of $\mathcal{L}_c$ is in Fig. 2.

Program fragments. A program fragment, $l : (t) : l'$, is a primitive statement t with a start label l and an end label l'. The label l identifies the left side of t, the *normal entry*, and l' identifies the right side of t, the *normal exit*. We also use $l_1 : (s_1; s_2) : l_3$ as an abbreviation for two fragments: $l_1 : (s_1) : l_2$ and $l_2 : (s_2) : l_3$, where l_2 is a new label. We use the symbol F for a set of fragments.

fragments	*Fragment*	$f ::= l : (t) : l'$
fragment sets	*FragSet*	$F ::= \{l_1 : (t_1) : l'_1,\ \ldots,\ l_n : (t_n) : l'_n\}$
assertions	*Assertion*	$p ::= \mathbf{true} \mid p_1 \vee p_2 \mid \neg p \mid re(e_1, \ldots, e_{ar(re)}) \mid \exists x.p$
label-continuation sets	*LContSet*	$\Psi ::= \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}$

Fig. 2. $\mathcal{L}_c$: Syntax

Assertions and label continuations. Assertions are meant to describe predicates on states. $\mathcal{L}_c$ can use any assertion language. We use first-order classical logic in this presentation (see Fig. 2). This assertion language is a superset of the language of boolean expressions. We omit conjunction and universal quantifiers since they can be defined by other constructors classically.

$\mathcal{L}_c$ is parametrized over a deduction system, $\mathcal{D}$, which derives true formulas in the assertion language. We leave the rules of $\mathcal{D}$ unspecified, and assume that its judgment is $\vdash_{\mathcal{D}} p$, which is read as that p is a true formula.

A label identifies a point in a program. To associate assertions with labels, $\mathcal{L}_c$ uses the notation, $l \triangleright p$, pronounced "l with p". In Hoare Logic, when an assertion p is associated with a label l in a verified program, then whenever the control of the program reaches l, the assertion p is true on the current state. In $\mathcal{L}_c$, we interpret $l \triangleright p$ in a different way: If $l \triangleright p$ is true in a program, then whenever p is satisfied, it is safe to continue from l (or, jump to l). Therefore, we call p a *precondition* of the label l, and call $l \triangleright p$ a *label continuation*. We use the symbol Ψ for a set of label continuations.

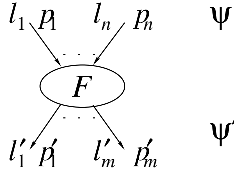
Form of specification. In $\mathcal{L}_c$, the judgment to specify properties of multiple-entry and multiple-exit program fragments has the syntax:

$$F ; \Psi' \vdash \Psi,$$

where F is a set of program fragments; Ψ' and Ψ are sets of label continuations. We next explain this judgment. Suppose

$$\Psi' = \{l'_1 \triangleright p'_1, \ldots, l'_m \triangleright p'_m\},\ \text{and}\ \Psi = \{l_1 \triangleright p_1, \ldots, l_n \triangleright p_n\}.$$

Labels $l'_1, \ldots, l'_m$ in Ψ' are exits of F, and $l_1, \ldots, l_n$ in Ψ are entries of F. The following graph depicts the relationship between F, Ψ, and Ψ':



With this relationship in mind, an informal interpretation of the judgment $F\,;\Psi' \;\vdash\; \Psi$ is as follows: for a set of fragments F, if it is safe to continue from any of the exit labels, provided that the associated assertion is true, then it is safe to continue from any of the entry labels, provided that the associated assertion is true. This interpretation draws conclusions on entry labels based on assumptions on exit labels. Note, however, this interpretation is simplified and the precise interpretation we will adopt for $F\,;\Psi' \;\vdash\; \Psi$ in Section 3 has an additional requirement: It takes at least one computation step from an entry to reach an exit. We ignore this issue for now and will come back to it.

Using this judgment, $\mathcal{L}_c$ provides rules for primitive statements. For example, Fig. 3 provides a rule for the assignment statement. In the assignment rule, the fragment, $l : (x := e) : l'$, has one entry, namely l, and one exit, namely l'. The assignment rule states that if it is safe to continue from the exit l', when p is true, then it is safe to continue from the entry l, when $p[e/x]$ is true. The reason why it is safe to continue from l can be informally established as follows: Suppose we start from l in an initial state where the next statement to execute is $x := e$ and $p[e/x]$ is true; the new state after the execution of the statement reaches the exit l', and based on the semantics of $x := e$, the assertion p is true; since we assume it is safe to continue from l', when p is true, the new state is safe to continue; hence, the initial state can safely continue from l, when $p[e/x]$ is true.

In Hoare Logic, the assignment rule is $\{p[e/x]\}\ x := e\ \{p\}$. This is essentially the same as the assignment rule in $\mathcal{L}_c$. In general, for any statement s that has only the normal entry and the normal exit, a Hoare triple $\{p\}s\{q\}$ has in $\mathcal{L}_c$ a corresponding judgment: $\{l : (s) : l'\}\,;\,\{l' \triangleright q\} \;\vdash\; \{l \triangleright p\}$.

But unlike Hoare triples, $F\,;\Psi' \;\vdash\; \Psi$ is a more general judgment, which is on multiple-entry and multiple-exit fragments. This capability is used in the rule for conditional-branch statements, **if** b **goto** l_1, in Fig. 3. A conditional-branch statement has two possible exits. Therefore, the if rule assumes two exit label continuations.

Composition rules. The strength of $\mathcal{L}_c$ is its composition rules. These rules can compose judgments on individual statements to form properties of the combined statement. By internalizing control flow of the combined statement, these composition rules allow modular reasoning.

Figure 3 shows $\mathcal{L}_c$'s composition rules. We illustrate these rules using the example in Fig. 4. The figure uses informal graphs, but they can be translated into formal syntax of $\mathcal{L}_c$ without much effort.

$\boxed{F\,;\,\Psi_1 \vdash \Psi_2}$

$$\frac{}{\{l : (x := e) : l'\}\,;\,\{l' \rhd p\} \vdash \{l \rhd p[e/x]\}}\ \textsf{assignment}$$

$$\frac{}{\{l : (\textbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p\} \vdash \{l \rhd p\}}\ \textsf{goto}$$

$$\frac{}{\{l : (\textbf{if}\ b\ \textbf{goto}\ l_1) : l'\}\,;\,\{l_1 \rhd p \wedge b, l' \rhd p \wedge \neg b\} \vdash \{l \rhd p\}}\ \textsf{if}$$

$$\frac{F_1\,;\,\Psi_1' \vdash \Psi_1 \quad F_2\,;\,\Psi_2' \vdash \Psi_2}{F_1 \cup F_2\,;\,\Psi_1' \cup \Psi_2' \vdash \Psi_1 \cup \Psi_2}\ \textsf{combine} \qquad \frac{F\,;\,\Psi' \cup \{l \rhd p\} \vdash \Psi \cup \{l \rhd p\}}{F\,;\,\Psi' \vdash \Psi \cup \{l \rhd p\}}\ \textsf{discharge}$$

$$\frac{\vdash \Psi_1' \Rightarrow \Psi_2' \quad F\,;\,\Psi_2' \vdash \Psi_2 \quad \vdash \Psi_2 \Rightarrow \Psi_1}{F\,;\,\Psi_1' \vdash \Psi_1}\ \textsf{weaken}$$

$\boxed{\vdash \Psi_1 \Rightarrow \Psi_2}$

$$\frac{m \geq n}{\vdash \{l_1 \rhd p_1, \ldots, l_m \rhd p_m\} \Rightarrow \{l_1 \rhd p_1, \ldots, l_n \rhd p_n\}}\ \textsf{s-width}$$

$$\frac{\vdash_{\mathcal{D}} p' \Rightarrow p}{\vdash \Psi \cup \{l \rhd p\} \Rightarrow \Psi \cup \{l \rhd p'\}}\ \textsf{s-depth}$$

Fig. 3. $\mathcal{L}_c$: Rules

Assume we already have two individual statements, depicted in the first column of Fig. 4, The first statement is an increment-by-one operation. If $x > 0$ before the statement, then after its completion $x > 0$ still holds. The second statement is **if** $x < 10$ **goto** l. It has one entry, but two exits. The entries and exits are associated with the appropriate assertions that are shown in the figure. The goal is to combine these two statements to form a property of the two-statement block. Notice that the block is effectively a repeat-until loop: it repeats incrementing x until x reaches 10. For this loop, our goal is to prove that if $x > 0$ before entering the block, then $x \geq 10$ after the completion of the block.

Figure 4 also presents the steps to derive the goal from the assumptions using $\mathcal{L}_c$'s composition rules.

In step 1, we use a rule called combine in Fig. 3. When combining two fragment sets, F_1 and F_2, the combine rule makes the union of the entries of F_1 and F_2 the entries of the combined fragment; the same goes for the exits. For the example in Fig. 4, since both statements have only one entry, we have two entries after the combine rule. Since the first statement has one exit, and the second statement has two exits, we have three exits after the combine rule.

After combining fragments, there may be some label that is both an entry and an exit. For example, the label l after the step 1 in Fig. 4 is both an entry and an exit. Furthermore, the entry and the exit for l carry the same assertion: $x > 0$. In such a case, the discharge rule in Fig. 3 can eliminate the label l as an exit. Formally, the discharge rule states that if some $l \rhd p$ appears on both the left and the right of the $\vdash$, then it can be removed from the left; Remember exits are on the left, so this rule removes an exit. The label l_1 is also both an entry and an

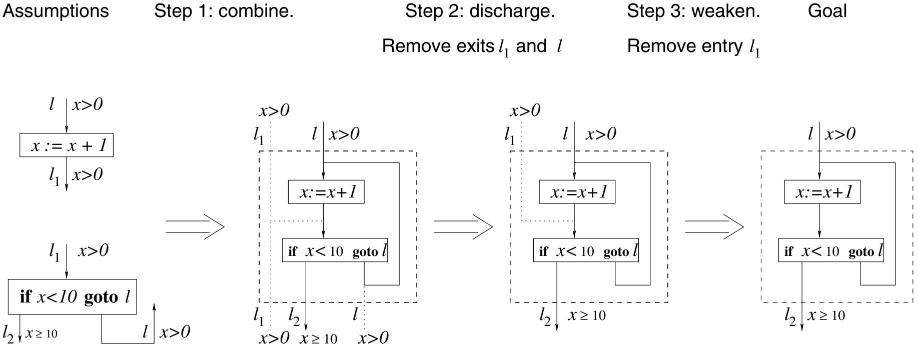


Fig. 4. An example to illustrate $\mathcal{L}_c$'s composition rules

exit, and the entry and the exit carry the same assertion. The discharge rule can remove l_1 as an exit as well. Therefore, the step 2 in Fig. 4 applies the discharge rule twice to remove both l and l_1 as exits. After this step, only one exit is left.

In the last step, we remove l_1 as an entry using the weaken rule. The weaken rule uses a relation between two sets of label continuations: $\vdash \Psi_1 \Rightarrow \Psi_2$, which is read as Ψ_1 is a stronger set of label continuations than Ψ_2.

The rule s-width in Fig. 3 states that a set of label continuations is stronger than its subset. Therefore, $\vdash \{l_1 \triangleright (x > 0), l \triangleright (x > 0)\} \Rightarrow \{l \triangleright (x > 0)\}$ is derivable. Using this result and the weaken rule, the step 3 in Fig. 4 removes the label l_1 as an entry.

After these steps, we have one entry and one exit left for the repeat-until loop, and we have proved the desired property for the loop.

One natural question to ask is which labels the logic should keep as entries. The example eliminates l_1 as an entry, while l remains. The reason is that the final goal tells what should be entries. In other scenarios, we may want to keep l_1 as an entry; for example, in cases when other fragments need to jump to l_1. This is possible in unstructured control flow even though l_1 points to the middle of a loop. In general, the logic $\mathcal{L}_c$ itself does not decide which entries to keep and needs extra information.

The example in Fig. 4 has used almost all composition rules, except for the s-depth rule. The s-depth rule states that a label continuation with a weaker precondition is stronger than a continuation with a stronger precondition. The rule is contravariant over the preconditions. An example of using this rule and the weaken rule is to derive $F\,;\Psi' \vdash \{l \triangleright p \wedge q\}$ from $F\,;\Psi' \vdash \{l \triangleright p\}$.

Deriving Hoare-logic rules. The composition rules in $\mathcal{L}_c$ can derive all Hoare-logic rules for common control-flow structures. We next show the derivation of the while rule. Assume that a while loop is implemented by the sequence in Equation (1) on page 81, which will be abbreviated by "**while** b **do** s". As we have mentioned, a Hoare triple $\{p\}s\{q\}$ corresponds to $\{l : (s) : l'\}\,; \{l' \triangleright q\} \vdash \{l \triangleright p\}$ in $\mathcal{L}_c$. With this correspondence, the derivation of the rule is:

$$\dfrac{\dfrac{\dfrac{\overline{(1)} \quad \{l_1 : (s) : l_2\}\,;\, \{l_2 \triangleright p\} \;\vdash\; \{l_1 \triangleright p \wedge b\} \quad \overline{(2)}\;\textsf{goto}}{\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l \triangleright p,\ l_1 \triangleright p \wedge b,\ l_2 \triangleright p,\ l' \triangleright p \wedge \neg b\} \;\vdash\; \{l \triangleright p,\ l_1 \triangleright p \wedge b,\ l_2 \triangleright p\}}\;\textsf{combine}}{\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l' \triangleright p \wedge \neg b\} \;\vdash\; \{l \triangleright p,\ l_1 \triangleright p \wedge b,\ l_2 \triangleright p\}}\;\textsf{discharge}}{\{l : (\textbf{while}\ b\ \textbf{do}\ s) : l'\}\,;\, \{l' \triangleright p \wedge \neg b\} \;\vdash\; \{l \triangleright p\}}\;\textsf{weaken}$$

where $(1) = \{l : (\textbf{if}\ \neg b\ \textbf{goto}\ l') : l_1\}\,;\, \{l' \triangleright p \wedge \neg b,\ l_1 \triangleright p \wedge b\} \;\vdash\; \{l \triangleright p\}$[1].
and $(2) = \{l_2 : (\textbf{goto}\ l) : l'\}\,;\, \{l \triangleright p\} \;\vdash\; \{l_2 \triangleright p\}$

In the same spirit, $\mathcal{L}_c$ can derive rules for many other control-flow structures, including sequential composition, repeat-until loops, if-then-else statements. More examples are in the thesis [16–chapter 2].

3 Semantics of $\mathcal{L}_c$

In this section, we develop a semantics for $\mathcal{L}_c$. We will show that a semantics based on pure continuations does not work. We adopt a semantics based on continuations together with approximations of counting computation steps.

3.1 Operational Semantics for the Language

First, we present an operational semantics for the imperative language in Fig. 1. The operational semantics assumes an interpretation $\oint$ of the primitive symbols in *OPSym* and *RSym* in the following way: *Val* is a nonempty domain; for each *op* in *OPSym*, its semantics, $\underline{op}$, is a function in $(\mathit{Val}^{ar(op)} \to \mathit{Val})$; for each *re* in *RSym*, $\underline{re}$ is a relation $\subset \mathit{Val}^{ar(re)}$, where $ar(op)$ is the arity of the operator.

A machine state is a triple, (pc, π, m): a program counter pc, which is an address; an instruction memory π, which maps addresses to primitive statements or to an **illegal** statement; a data memory m, which maps variables to values. Figure 5 lists the relevant semantic domains.

Before presenting the operational semantics, we introduce some notation. For a state σ, the notation $\mathrm{control}(\sigma)$, $\mathrm{i_of}(\sigma)$, and $\mathrm{m_of}(\sigma)$ projects σ into its program counter, instruction memory, and data memory, respectively. For a mapping m, the notation $m[x \mapsto v]$ denotes a new mapping that maps x to v and leaves other slots unchanged.

The operational semantics for the language is presented in Fig. 6 as a step relation $\sigma \mapsto_\theta \sigma'$ that executes the statement pointed by the program counter. The operational semantics is conventional, except that it is parametrized over a label map $\theta \in \mathit{LMap}$, which maps abstract labels to concrete addresses. When the next statement to execute is **goto** l, the control is changed to $\theta(l)$.

---

[1] The judgment is derived from the if rule and the weaken rule, assuming that $\vdash_{\mathfrak{D}} p \wedge \neg\neg b \Rightarrow p \wedge b$.

Name	Domain Construction
values, v	Val is a nonempty domain
addresses, n	$Addr = \mathbb{N}$
instr. memories, π	$IM = Addr \to PrimStmt \cup \{\textbf{illegal}\}$
data memories, m	$DM = Var \to Val$
states, σ	$\Sigma = Addr \times IM \times DM$
label maps, θ	$LMap = Label \to Addr$

where $\mathbb{N}$ is the domain of natural numbers.

Fig. 5. Semantic domains

$(\mathrm{pc}, \pi, m) \mapsto_\theta \sigma$ where	
if $\pi(\mathrm{pc}) =$	then $\sigma =$
$x := e$	$(\mathrm{pc}+1, \pi, m[x \mapsto \mathcal{V}[\![e]\!]\,m])$
goto l	$(\theta(l), \pi, m)$
if b **goto** l	$\begin{cases} (\theta(l), \pi, m) & \text{if } \mathcal{B}[\![b]\!]\,m = \mathrm{tt} \\ (\mathrm{pc}+1, \pi, m) & \text{otherwise} \end{cases}$

where $\mathcal{V} : Exp \to DM \to Val$, and $\mathcal{B} : BExp \to DM \to \{\mathrm{tt}, \mathrm{ff}\}$.

Their definitions are

$$\mathcal{V}[\![x]\!]\,m \triangleq m\,[x] \qquad \mathcal{V}[\![op(e_1, \ldots, e_{ar(op)})]\!]\,m \triangleq \underline{op}(\mathcal{V}[\![e_1]\!]\,m, \ldots, \mathcal{V}[\![e_{ar(op)}]\!]\,m).$$

$$\mathcal{B}[\![\mathbf{true}]\!]\,m \triangleq \mathrm{tt} \qquad \mathcal{B}[\![b_1 \vee b_2]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}[\![b_1]\!]\,m = \mathrm{tt} \text{ or } \mathcal{B}[\![b_2]\!]\,m = \mathrm{tt} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![\neg b]\!]\,m \triangleq \begin{cases} \mathrm{tt} & \text{if } \mathcal{B}[\![b]\!]\,m = \mathrm{ff} \\ \mathrm{ff} & \text{otherwise} \end{cases}$$

$$\mathcal{B}[\![re(e_1, \ldots, e_{ar(re)})]\!]\,m \triangleq \begin{cases} \mathrm{tt} \text{ if } \langle \mathcal{V}[\![e_1]\!]\,m, \ldots, \mathcal{V}[\![e_{ar(re)}]\!]\,m \rangle \in \underline{re} \\ \mathrm{ff} \text{ otherwise} \end{cases}$$

Fig. 6. Operational semantics of the language in Fig. 1

In the operational semantics, if the current statement in a state σ is an **illegal** statement, then σ has no next state to step to; such a state is called a stuck state. If a state σ will not reach a stuck state within k steps, it is *safe for k steps*:

$$\mathrm{safe_state}(\sigma, k) \;\triangleq\; \forall \sigma' \in \Sigma. \forall j < k.\;\; \sigma \mapsto_\theta^j \sigma' \;\Rightarrow\; \exists \sigma''.\; \sigma' \mapsto_\theta \sigma'',$$

where $\mapsto_\theta^j$ denotes j steps being taken.

3.2 Semantics of $\mathcal{L}_c$

The semantics of $\mathcal{L}_c$ is centered on an interpretation of the judgment $F\,; \Psi' \vdash \Psi$. We have discussed an informal interpretation: for the set of fragments F, if Ψ' is true, then Ψ is true; a label-continuation set Ψ being true means it is safe to continue from any label in Ψ, provided that the associated assertion is true. However, this interpretation is too naive, since it cannot justify the discharge rule. When both Ψ' and Ψ in the discharge rule are empty sets, the rule becomes

$$\frac{F\,;\,\{l \triangleright p\} \vdash \{l \triangleright p\}}{F\,;\,\emptyset \vdash \{l \triangleright p\}}$$

According to the informal interpretation, the above rule is like stating "from $l \triangleright p \Rightarrow l \triangleright p$, derive $l \triangleright p$", which is clearly unsound.

The problem is not that $\mathcal{L}_c$ is intrinsically unsound, but that the interpretation is too weak to utilize invariants implicitly in $\mathcal{L}_c$. The interpretation that we adopt is a stronger one. The basic idea is based on a notion of label continuations being *approximately* true. The judgment $F\,;\,\Psi' \vdash \Psi$ is interpreted as, by assuming the truth of Ψ' at a lower approximation, Ψ is true at a higher approximation. In this inductive interpretation, Ψ' and Ψ are treated differently, and it allows the discharge rule to be justified by induction.

Appel and McAllester proposed the indexed model [17], where all predicates are approximated by *counting computation steps*. Our own work [18] used the indexed model to construct a semantic model for a typed assembly language. Next, we will adopt the idea of approximation by counting computation steps from the indexed model to develop a semantics for $\mathcal{L}_c$.

Label continuations being approximately true. We first introduce a semantic function, $\mathcal{A} : \mathit{Assertion} \to \mathit{DM} \to \{\mathrm{tt}, \mathrm{ff}\}$, which gives a meaning to assertions:

$$\mathcal{A}\,[\![\exists x.p]\!]\,m \triangleq \begin{cases} \mathrm{tt} \text{ if } \exists d \in \mathit{Val}.\ \mathcal{A}\,[\![p[d/x]]\!]\,m = \mathrm{tt} \\ \mathrm{ff} \text{ otherwise.} \end{cases}$$

The definition of "$\mathcal{A}\,[\![p]\!]\,m$" on other cases of p is the same as the definition of $\mathcal{B}$ (in Fig. 6) except every occurrence of $\mathcal{B}$ is replaced by $\mathcal{A}$.

Next, we present a notion, $\sigma; \theta \models_k l \triangleright p$, to mean that a label continuation $l \triangleright p$ is k-approximately true in state σ relative to a label map θ:

$$\begin{aligned} &\sigma; \theta \models_k l \triangleright p \ \triangleq \\ &\quad \forall \sigma' \in \Sigma.\ \sigma \mapsto^*_\theta \sigma' \ \wedge\ \mathrm{control}(\sigma') = \theta(l) \ \wedge\ \mathcal{A}\,[\![p]\!]\,(\mathrm{m_of}(\sigma')) = \mathrm{tt} \\ &\qquad\qquad \Rightarrow\ \mathrm{safe_state}(\sigma', k) \end{aligned} \tag{3}$$

where $\mapsto^*_\theta$ denotes multiple steps being taken.

There are several points that need to be clarified about the definition. First, by this definition, $l \triangleright p$ being a true label continuation in σ to approximation k means that the state is safe to execute for k steps. In other words, the state will not get stuck within k steps.

Second, the definition is relative to a label map θ, which is used to translate the abstract label l to its concrete address.

Last, the definition quantifies over all future states σ' that σ can step to (including σ itself). The reason is that if $\sigma; \theta \models_k l \triangleright p$, provided that p is satisfied, it should be safe to continue from location l, not just now, but also in the future. In other words, if $l \triangleright p$ is true in the current state, it should also be true in all future states. Therefore, the definition of $\sigma; \theta \models_k l \triangleright p$ has to satisfy the following lemma:

Lemma 1. *If $\sigma \mapsto^*_\theta \sigma'$, and $\sigma; \theta \models_k l \triangleright p$, then $\sigma'; \theta \models_k l \triangleright p$.*

By quantifying over all future states, the definition of $\sigma; \theta \models_k l \triangleright p$ satisfies the above lemma. On this aspect, the semantics of $\sigma; \theta \models_k l \triangleright p$ is similar to the

Kripke model [19–Ch 2.5] of intuitionistic logic: Knowledge is preserved from current states to future states.

The semantics of a single label continuation is then extended to a set of label continuations: $\sigma;\theta \models_k \Psi \;\triangleq\; \forall(l \triangleright p) \in \Psi.\ \sigma;\theta \models_k l \triangleright p$

Loading statements. The predicate loaded(F, π, θ) describes the loading of a fragment set F into an instruction memory π with respect to a label mapping θ:

$$\text{loaded}(F, \pi, \theta) \;\triangleq\; \forall(l : (t) : l') \in F.\ \pi(\theta(l)) = t \;\wedge\; \theta(l') = \theta(l) + 1.$$

Note that some θ are not valid with respect to F. For example, if $F = \{l : (x := 1) : l'\}$, and θ maps l to address 100, then to be consistent θ has to map l' to the address 101. This is the reason why the definition requires[2] that $\theta(l') = \theta(l) + 1$.

Semantics of the judgment $F\,;\Psi' \vdash \Psi$. We define a relation, $F\,;\Psi' \models \Psi$, which is the semantic modeling of $F\,;\Psi' \vdash \Psi$.

$$\begin{aligned}
&F\,;\Psi' \models \Psi \triangleq \\
&\quad \forall \sigma \in \Sigma, \theta \in \mathit{LMap}.\ \text{loaded}(F, \text{i_of}(\sigma), \theta) \Rightarrow \\
&\quad\quad \forall k \in \mathbb{N}.\ \big(\sigma;\theta \models_k \Psi' \;\Rightarrow\; \sigma;\theta \models_{k+1} \Psi\big).
\end{aligned}$$

The definition quantifies over all label maps θ and all states σ such that F is loaded in the state with respect to θ. It derives the truth of Ψ to approximation $k + 1$, from the truth of Ψ' to approximation k. In other words, if it is safe to continue from any of the labels in Ψ', provided that the associated assertion is true, for some number k of computation steps, then it is safe to continue from any of the labels in Ψ, provided that the associated assertion is true, for $k + 1$ computation steps. This inductive definition allows the discharge rule to be proved by induction over k.

We have given $F\,;\Psi' \models \Psi$ a strong definition. But the question is what about rules other than the discharge rule. Do they support such a strong semantics? The answer is yes for $\mathcal{L}_c$, because of one implicit invariant—for any judgment $F\,;\Psi' \vdash \Psi$ that is derivable, it takes at least one computation step from labels in Ψ to reach labels in Ψ'. Or, it takes at least one step from entries of F to reach an exit of F. Because of this invariant, although it is safe to continue from exit labels only for k steps, we can still show that it is safe to continue from entry labels for $k + 1$ steps.

Finally, since $\mathcal{L}_c$ also contains rules for deriving $\vdash \Psi \Rightarrow \Psi'$ and $\vdash_{\mathfrak{D}} p$, we define relations, $\models \Psi \Rightarrow \Psi'$ and $\models p$, to model their meanings, respectively.

$$\begin{aligned}
&\models \Psi \Rightarrow \Psi' \triangleq \forall \sigma \in \Sigma, \theta \in \mathit{LMap}, k \in \mathbb{N}.\ (\sigma;\theta \models_k \Psi) \Rightarrow (\sigma;\theta \models_k \Psi') \\
&\models p \qquad\quad\; \triangleq \forall m \in \mathit{DM}.\ \mathcal{A}\,[\![p]\!]\,m = \text{tt}
\end{aligned}$$

[2] There is a simplification. The definition in the thesis [16] also requires that θ does not map exit labels to addresses occupied by F; otherwise, the exit label would not be a “true” exit label.

Soundness and completeness. Based on the semantics we have developed, we next present soundness and completeness theorems for $\mathcal{L}_c$. Due to space limit, we only informally discuss related concepts and cannot present detailed proofs; they can be found in the thesis [16].

As a start, since $\mathcal{L}_c$ is parametrized by a deduction system $\mathcal{D}$, which derives formulas in the assertion language, it is necessary to assume properties of $\mathcal{D}$ before proving properties of $\mathcal{L}_c$: If $\vdash_{\mathcal{D}} p \Rightarrow \models p$, for any p, then $\mathcal{D}$ is *sound*; if $\models p \Rightarrow \vdash_{\mathcal{D}} p$, for any p, then $\mathcal{D}$ is *complete*.

Next, we present the soundness and completeness theorems of $\mathcal{L}_c$.

Theorem 1. *(Soundness) Assume $\mathcal{D}$ is sound. If $F\,;\Psi \vdash \Psi'$, then $F\,;\Psi \models \Psi'$.*

The proof is by induction over the derivation of $F\,;\Psi \vdash \Psi'$. The most interesting case is the proof of the discharge rule, which is proved by induction over the number of future computation steps k.

Theorem 2. *(Completeness.)*
Assume $\mathcal{D}$ is complete and Assertion is expressive relative to $\oint$. Assume Assertion is negatively testable by the statement language. Assume (F, Ψ', Ψ) is normal. If $F\,;\Psi' \models \Psi$, then $F\,;\Psi' \vdash \Psi$.

We informally explain the meanings of expressiveness, *Assertion* being negatively testable, and (F, Ψ', Ψ) being normal below; their precise definitions are in the thesis [16]. As pointed out by Cook [20], a program logic can fail to be complete, if the assertion language is not powerful enough to express invariants for loops in a program. Therefore, the completeness theorem assumes that the assertion language is *expressive*. Also, the theorem assumes that the assertion language is *negatively testable* by the statement language. It means that for any assertion p, there is a sequence of statements that terminates when p is false and diverges when p is true. The triple (F, Ψ', Ψ) being normal means that any label is defined in F at most once; it includes also other sanity requirements on Ψ' and Ψ.

4 Implementation in FPCC

This work is a part of the Foundational Proof-Carrying Code (FPCC) project [4] at Princeton. FPCC verifies memory safety of machine code from the smallest possible set of axioms—machine semantics plus logic. The safety proof is developed in two stages. First, we design a type system at the machine-code level. Machine code is type checked in the type system and thus a typing derivation is a safety witness. In the second stage, we prove the soundness theorem for the type system: If machine code type checks, it is memory safe. This proof is developed with respect to machine semantics plus logic, and is machine checked. The typing derivation composed with the soundness proof is the safety proof of the machine code. The major research problem of the FPCC project is to prove the soundness of our type system—a low-level typed assembly language (LTAL [21]). LTAL can check the memory-safety of SPARC machine code that is generated from our ML compiler.

When proving the soundness of LTAL, we found it is easier to have an intermediate calculus to aid the proving process, because having a simple soundness proof was not LTAL's design goal. We first prove the intermediate calculus is sound from logic plus machine semantics. Then we prove LTAL is sound based on the lemmas provided by the intermediate calculus.

The intermediate calculus in the FPCC project is $\mathcal{L}_c$, together with a type theory [22] as the assertion language. By encoding on top of SPARC machine the semantics of $\mathcal{L}_c$, which we have presented, we have proved that $\mathcal{L}_c$ is sound with machine-checked proofs in Twelf. Then, we prove that LTAL is sound from the lemmas provided by $\mathcal{L}_c$.

The first author's thesis [16–chapter 3] covers the step from $\mathcal{L}_c$ to LTAL in great detail. Here we only discuss a point about control-flow structures in machine code. The simple language in Section 2 on which we presented $\mathcal{L}_c$ lacks indirect jumps, pc-relative jumps, and procedure calls. Our implementation on SPARC handles pc-relative jumps, and handles indirect jumps using first-class continuation types in the assertion language. These features will not affect the soundness result of $\mathcal{L}_c$, as our implementation has shown. However, we have not investigated the impact of indirect jumps to the completeness result, which is unnecessary for the FPCC project. We have not modeled procedure call-and-return—since our compiler uses continuation-passing style, continuation calls and continuation-passing suffice. Procedure calls, if needed, could be handled by following the work of Benton [13].

5 Conclusion and Future Work

Previous program logics for goto statements are too weak to modularly reason about program fragments with multiple entries and multiple exits. We have presented $\mathcal{L}_c$, which needs only local information to reason about a program fragment and compose program fragments in an elegant way. $\mathcal{L}_c$ is not only useful for reasoning about unstructured control flow in machine languages, but also useful for deriving rules for common control-flow structures. We have also presented for $\mathcal{L}_c$ a semantics, based on which the soundness and completeness theorems are formally proved. We have implemented $\mathcal{L}_c$ on top of SPARC machine language. The implementation has been embedded into the Foundational Proof-Carrying Code Project to produce memory-safety proofs for machine-language programs.

One possible future extension is to combine this work with modules, to produce a module system with simple composition rules, and with a semantics based on counting computation steps.

References

1. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the Association for Computing Machinery **12** (1969) 578–580
2. Necula, G.: Proof-carrying code. In: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, ACM Press (1997) 106–119

3. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. on Programming Languages and Systems **21** (1999) 527–568
4. Appel, A.W.: Foundational proof-carrying code. In: Symposium on Logic in Computer Science (LICS '01), IEEE (2001) 247–258
5. Clint, M., Hoare, C.A.R.: Program proving: Jumps and functions. Acta Informatica (1972) 214–224
6. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta Informatica **7** (1977) 357–360
7. Arbib, M., Alagic, S.: Proof rules for gotos. Acta Informatica **11** (1979) 139–148
8. de Bruin, A.: Goto statements: Semantics and deduction systems. Acta Informatica **15** (1981) 385–424
9. O'Donnell, M.J.: A critique of the foundations of hoare style programming logics. Communications of the Association for Computing Machinery **25** (1982) 927–935
10. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposia in Applied Mathematics, Providence, Rhode Island (1967) 19–32
11. Cardelli, L.: Program fragments, linking, and modularization. In: 24th ACM Symposium on Principles of Programming Languages. (1997) 266–277
12. Glew, N., Morrisett, G.: Type-safe linking and modular assembly language. In: 26th ACM Symposium on Principles of Programming Languages. (1999) 250–261
13. Benton, N.: A typed, compositional logic for a stack-based abstract machine. In: 3rd Asian Symposium on Programming Languages and Systems. (2005)
14. Ni, Z., Shao, Z.: Certified assembly programming with embedded code pointers. In: 33rd ACM Symposium on Principles of Programming Languages. (2006) To appear.
15. Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. In: Proceedings of the Second Workshop on Structured Operational Semantics (SOS'05). (2005)
16. Tan, G.: A Compositional Logic for Control Flow and its Application in Foundational Proof-Carrying Code. PhD thesis, Princeton University (2005)
17. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. on Programming Languages and Systems **23** (2001) 657–683
18. Tan, G., Appel, A.W., Swadi, K.N., Wu, D.: Construction of a semantic model for a typed assembly language. In: Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04). (2004) 30–43
19. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Available as DIKU Rapport 98/14 (1998)
20. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing **7** (1978) 70–90
21. Chen, J., Wu, D., Appel, A.W., Fang, H.: A provably sound TAL for back-end optimization. In: ACM Conference on Programming Language Design and Implementation. (2003) 208–219
22. Swadi, K.N.: Typed Machine Language. PhD thesis, Princeton University (2003)

Detecting Non-cyclicity by Abstract Compilation into Boolean Functions

Stefano Rossignoli and Fausto Spoto

Dipartimento di Informatica, Università di Verona,
Strada Le Grazie, 15, 37134 Verona, Italy
`stefano.rossignoli@students.univr.it`,
`fausto.spoto@univr.it`

Abstract. Programming languages such as C, C++ and Java bind variables to dynamically-allocated data-structures held in memory. This lets programs build cyclical data at run-time, which complicates termination analysis and garbage collection. It is hence desirable to spot those variables which are only bound to non-cyclical data at run-time. We solve this problem by using abstract interpretation to define the abstract domain NC representing those variables. We relate NC through a Galois insertion to the concrete domain of program states. Hence NC is not redundant. We define a correct abstract denotational semantics over NC, which uses preliminary sharing information between variables to get more precise results. We apply it to a simple example of analysis. We use a Boolean representation for the abstract denotations over NC, which leads to an efficient implementation in terms of binary decision diagrams and to the elegant and efficient use of abstract compilation.

1 Introduction

Programming languages such as C, C++ and Java allocate dynamic data-structures on the heap. These data-structures might contain cycles, which hinder termination analysis and complicate garbage collection.

Consider the classes in Figure 1, in the syntax of Section 4. We observe here that `with` introduces local variables, and variable *out* holds the return value of a method. These classes implement a list of subscriptions to a service, such as cable television. Each subscription refers to a person. Some subscriptions come from abroad, and have a higher monthly cost. The method `foreign` over a list of subscriptions builds a new list, containing only the foreign ones.

If, in a call such as *l1*:=*l2*.`foreign`(), we know that *l2* is bound to a non-cyclical data-structure (from now on, if *l2* is a *non-cyclical variable*), then

1. techniques for termination analysis, similar to those for logic programs [1], based on decreasing *norms*, can prove that the call terminates; non-cyclicity plays here the role of the occur-check in logic programming;
2. a (single pass) *reference counting* garbage collector can be applied to *l1* and *l2* when they go out of scope, since they do not lead to cycles. This is faster than a (two passes) *mark and sweep* collector and has smaller pause times.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 95–110, 2006.
© Springer-Verlag Berlin Heidelberg 2006

```
class Object {}
class Person extends Object { int age; boolean sex; }
class Subs extends Object {
  Person subscriber; int numOfChannels; Subs next;
  int monthlyCost() { out := numOfChannels / 2 } // in euros
  ForeignSubs foreign() with temp:Subs {
    temp = this.next;
    if (temp = null) then {} else out := temp.foreign() }
}
class ForeignSubs extends Subs {
  int monthlyCost() { out := numOfChannels * 2 } // more expensive
  ForeignSubs foreign() with temp:Subs, sub:Person {
    sub := this.subscriber;  out := new ForeignSubs;  // program point *
    temp := this.next;
    if (temp = null) then {} else out.next := temp.foreign();
    out.subscriber := sub;  out.numOfChannels := this.numOfChannels }
}
```

Fig. 1. Our running example: a list of subscriptions to a service

Hence detecting the non-cyclical variables helps program verification and optimise the run-time support. One *might* derive non-cyclicity by abstracting some *shape analyses* [10] and some *alias* or *sharing* analyses which build a graph-representation of the run-time heap of the system [9]. In this paper, we follow a different approach. We use abstract interpretation [5] to define a domain NC for non-cyclicity, more abstract than graphs. NC is the same property we want to observe *i.e.,* the set of non-cyclical variables. Because of the simplicity of NC,

– a Galois *insertion* relates NC to the concrete domain of program states. Hence NC is not redundant [5];
– ours is a denotational and hence completely *relational* analysis which denotes each method with its *interpretation i.e.,* its abstract input/output behaviour;
– we represent these behaviours through Boolean functions, which can be efficiently implemented through *binary decision diagrams* [4];
– we use *abstract compilation* [6] into Boolean functions, an elegant and efficient implementation of abstract interpretation.

Our analysis is possibly less precise than abstractions of graph-based analyses (as *some* shape, alias or sharing analyses), which keep explicit information on the names of the fields of the objects in the heap. This information is abstracted away in NC. But those analyses do not enjoy all the properties stated above for NC, are much more concrete than NC and have never been actually abstracted to non-cyclicity analysis. The precision of NC can be significantly improved by using some preliminary sharing information between program variables [8].

The paper is organised as follows. Section 2 presents our analysis. Section 3 reports the preliminaries. Section 4 gives syntax and semantics of our simple object-oriented language. Section 5 defines the domain NC. Section 6 gives a

denotational semantics over NC. Section 7 represents denotations over NC as Boolean functions. Section 8 concludes. Proofs are in [7].

2 Overview of the Analysis

We show here informally how abstract compilation over Boolean functions works for our non-cyclicity analysis. This will be formalised in subsequent sections.

Consider the `foreign` method of class `Subs` in Figure 1. We want to prove that its result is always non-cyclical. Our analysis starts by *compiling* all methods into Boolean functions, by following the *compilation rules* in Figure 7. The result, for the method `Subs.foreign`, is in Figure 2. Each program variable v is split into its input and output version $\check{v}$ and $\hat{v}$. For instance, the compilation of $s = (temp{:=}this.\mathtt{next})$ in Figure 1 is the formula ϕ_1 in Figure 2, stating that if *this* is non-cyclical before the execution of s then *temp* is non-cyclical after the execution of s (*i.e.,* $\check{this} \to \hat{temp}$) and that the non-cyclicity of *this* and *out* is not affected by s (*i.e.,* $(\check{this} \to \hat{this}) \wedge (\check{out} \to \hat{out})$). The `then` and `else` branches of the conditional are compiled into ϕ_2 and ϕ_3, respectively, and the conditional itself into $\phi_2 \vee \phi_3$. Formula ϕ_2 states that *temp* is *null* and hence non-cyclical at the beginning of the `then` branch (*i.e.,* $\check{temp}$), which moreover does not change the cyclicity of any variable (*i.e.,* $(\check{this} \to \hat{this}) \wedge (\check{out} \to \hat{out}) \wedge (\check{temp} \to \hat{temp})$). Formula ϕ_3 is rather complex since it is the compilation of a method call. We will discuss it in subsequent sections. Here, just note that it refers to unknown formulas $I(\mathtt{Subs.foreign})$ and $I(\mathtt{ForeignSubs.foreign})$, the *current interpretation* of those methods. They express what we already know about the abstract behaviour of those methods. Variable *res* holds the value of the method call expression. The operation $\circ$ on Boolean functions corresponds to sequential composition of denotations and is defined in Section 7.

```
ForeignSubs foreign() with temp:Subs {
```

$$
\begin{array}{l}
[(\check{this} \to \hat{temp}) \wedge (\check{this} \to \hat{this}) \wedge (\check{out} \to \hat{out})] \circ \quad \}\phi_1 \\
\Big[[\check{temp} \wedge (\check{this} \to \hat{this}) \wedge (\check{out} \to \hat{out}) \wedge (\check{temp} \to \hat{temp})] \vee \quad \}\phi_2 \\
\Big\{\Big(\Big((\check{temp} \to \hat{this}) \circ \begin{pmatrix} I(\mathtt{Subs.foreign}) \vee \\ I(\mathtt{ForeignSubs.foreign}) \end{pmatrix} \circ (\check{out} \to \hat{res})\Big) \wedge (\check{out} \to \hat{out})\Big) \\
\quad \circ \Big((\check{res} \to \hat{out}) \wedge (\check{this} \to \hat{this}) \wedge (\check{temp} \to \hat{temp})\Big)\Big]\Big\} \quad \Big\}\phi_3
\end{array}
$$

```
}
```

Fig. 2. The Boolean compilation of the `Subs.foreign` method in Figure 1

After abstract compilation, our analysis builds a *bottom* interpretation I which uses to evaluate the body (formula) of each method. This yields a new interpretation. This process is iterated until the fixpoint, which is the result of the analysis. It expresses a completely relational input/output behaviour for the methods [5, 3]. In our case, the method in Figure 2 is interpreted by the formula $\hat{out}$ *i.e.,* its return value is always non-cyclical (Example 16).

3 Preliminaries

A total (partial) function f is denoted by $\mapsto$ ($\rightarrow$). The *domain* (*codomain*) of f is $\mathrm{dom}(f)$ ($\mathrm{rng}(f)$). We denote by $[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ the function f where $dom(f) = \{v_1, \ldots, v_n\}$ and $f(v_i) = t_i$ for $i = 1, \ldots, n$. Its *update* is $f[w_1 \mapsto d_1, \ldots, w_m \mapsto d_m]$, where the domain may be enlarged. By $f|_s$ ($f|_{-s}$) we denote the *restriction* of f to $s \subseteq \mathrm{dom}(f)$ (to $\mathrm{dom}(f) \setminus s$). If $f(x) = x$ then x is a *fixpoint* of f. The composition $f \circ g$ of functions f and g is such that $(f \circ g)(x) = g(f(x))$ so that we often denote it as gf. The components of a *pair* are separated by $\star$. A definition of S such as $S = a \star b$, with a and b meta-variables, silently defines the pair selectors $s.a$ and $s.b$ for $s \in S$.

A *poset* $S \star \leq$ is a set S with a reflexive, transitive and antisymmetric relation $\leq$. Given $s \in S$, we define $\downarrow s = \{s' \in S \mid s' \leq s\}$. If $C \star \leq$ and $A \star \preceq$ are posets (the *concrete* and the *abstract* domain), a *Galois connection* [5] is a pair of monotonic maps $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ (*i.e.,* $\alpha(c_1) \preceq \alpha(c_2)$ if $c_1 \leq c_2$, similarly for γ) such that $\gamma\alpha$ is extensive ($c \leq \gamma\alpha(c)$ for any $c \in C$) and $\alpha\gamma$ is reductive ($\alpha\gamma(a) \leq a$ for any $a \in A$). It is a *Galois insertion* if $\alpha\gamma$ is the identity map *i.e.,* if A contains no redundancy *i.e.,* if γ is one-to-one. An abstract operator $\hat{f} : A^n \rightarrow A$ is *correct w.r.t.* a concrete $f : C^n \rightarrow C$ if $\alpha f \gamma \preceq \hat{f}$.

4 Our Simple Object-Oriented Language

Syntax. Variables are typed and bound to values. We do not consider primitive types since they cannot be used to build cycles.

Definition 1. *A program has a set of* variables $\mathcal{V}$ *(including* res, out, $this$*) and a finite set of* classes *(or* types*)* $\mathcal{K}$ *ordered by a* subclass relation $\leq$. *A* type environment *specifies the types of a finite set of variables. It is any element of the set* $TypEnv = \{\tau : \mathcal{V} \rightarrow \mathcal{K} \mid \mathrm{dom}(\tau) \text{ is finite}\}$. *Later on,* τ *will stand for a type environment. Type environments describe both the variables in scope at a given program point and the fields of a given class* $\kappa \in \mathcal{K}$*, written as* $F(\kappa)$*.*

Example 1. In Figure 1 we have $\mathcal{K} = \{\mathtt{Object}, \mathtt{Person}, \mathtt{Subs}, \mathtt{ForeignSubs}\}$, where `Object` is the top of the hierarchy and `ForeignSubs` $\leq$ `Subs`. Since we are not interested in primitive types, we have $F(\mathtt{Object}) = F(\mathtt{Person}) = []$ and $F(\mathtt{Subs}) = F(\mathtt{ForeignSubs}) = [\mathtt{subscriber} \mapsto \mathtt{Person}, \mathtt{next} \mapsto \mathtt{Subs}]$.

Expressions and commands are normalised versions of Java's. Only distinct variables can be actual parameters of a method call; leftvalues are only a variable or the field of a variable; conditionals only check equality or nullness of variables; loops are implemented through recursion. We can relax these simplifying assumptions without affecting subsequent results. It is significant that we allow downwards casts, since reachability (Definition 6) depends from their presence.

Definition 2. Expressions[1] *and* commands *are*

$$exp ::= \texttt{null}\ \kappa \mid \texttt{new}\ \kappa \mid v \mid v.\texttt{f} \mid (\kappa)v \mid v.\texttt{m}(v_1, \ldots, v_n)$$
$$com ::= v := exp \mid v.\texttt{f} := exp \mid \{com; \cdots ;com\}$$
$$\mid \texttt{if}\ v = w\ \texttt{then}\ com\ \texttt{else}\ com \mid \texttt{if}\ v = \texttt{null}\ \texttt{then}\ com\ \texttt{else}\ com$$

where $\kappa \in \mathcal{K}$ *and* $v, w, v_1, \ldots, v_n \in \mathcal{V}$ *are distinct.*

Each method $\kappa.\texttt{m}$ *is* defined *in class* κ *as* $\kappa_0\ \texttt{m}(w_1 : \kappa_1, \ldots, w_n : \kappa_n)\ \texttt{with}\ w_{n+1} : \kappa_{n+1}, \ldots, w_{n+m} : \kappa_{n+m}\ \texttt{is}\ com$*, where* $w_1, \ldots, w_n, w_{n+1}, \ldots, w_{n+m} \in \mathcal{V}$ *are distinct, not in* $\{out, res, this\}$ *and have* static type $\kappa_1, \ldots, \kappa_n, \kappa_{n+1}, \ldots, \kappa_{n+m} \in \mathcal{K}$*, respectively. Variables* $w_1, \ldots, w_n$ *are the* formal parameters *of the method,* $w_{n+1}, \ldots, w_{n+m}$ *are its* local variables*. The method also uses a variable out of type* κ_0 *to store its* return value*. We let* $body(\kappa.\texttt{m}) = com$*,* $returnType(\kappa.\texttt{m}) = \kappa_0$*,* $input(\kappa.\texttt{m}) = [this \mapsto \kappa, w_1 \mapsto \kappa_1, \ldots, w_n \mapsto \kappa_n]$*,* $output(\kappa.\texttt{m}) = [out \mapsto \kappa_0]$*,* $locals(\kappa.\texttt{m}) = [w_{n+1} \mapsto \kappa_{n+1}, \ldots, w_{n+m} \mapsto w_{n+m}]$ *and* $scope(\kappa.\texttt{m}) = input(\kappa.\texttt{m}) \cup output(\kappa.\texttt{m}) \cup locals(\kappa.\texttt{m})$*.*

Example 2. For `ForeignSubs.foreign` in Figure 1 (just `foreign` below) we have $input(\texttt{foreign}) = [this \mapsto \texttt{ForeignSubs}]$, $output(\texttt{foreign}) = [out \mapsto \texttt{ForeignSubs}]$, $locals(\texttt{foreign}) = [sub \mapsto \texttt{Person}, temp \mapsto \texttt{Subs}]$.

Our language is strongly typed *i.e.,* expressions *exp* have a static (compile-time) type $type_\tau(exp)$ in τ, consistent with their run-time values (see [7]).

Semantics. We use a denotational semantics, hence compositional, in the style of [11]. However, we use more complex states, which include a heap. By using a denotational semantics, our states contain only a single frame, rather than an activation stack of frames. A method call is hence resolved by *plugging* the interpretation of the method (Definition 5) in its calling context. This is standard in denotational semantics and has been used for years in logic programming [3].

A frame binds variables (identifiers) to locations or *null*. A memory binds such locations to objects, which contain a class tag and the frame for their fields.

Definition 3. *Let Loc be an infinite set of* locations*. We define* frames*,* objects *and* memories *as* $Frame_\tau = \{\phi \mid \phi \in \mathrm{dom}(\tau) \mapsto Loc \cup \{null\}\}$*,* $Obj = \{\kappa \star \phi \mid \kappa \in \mathcal{K},\ \phi \in Frame_{F(\kappa)}\}$ *and* $Memory = \{\mu \in Loc \to Obj \mid \mathrm{dom}(\mu)\ \textit{is finite}\}$*. A new object of class* κ *is* $new(\kappa) = \kappa \star \phi$*, with* $\phi(f) = null$ *for each* $f \in F(\kappa)$*.*

Example 3. Figure 3 shows a frame ϕ and a memory μ. Different occurrences of the same location are linked. For instance, variable *this* is bound to location l_1 and $\mu(l_1)$ is a `ForeignSubs` object. Objects are shown as boxes in μ with a class tag and a local frame mapping fields to locations or *null*. The state in Figure 3 might be the current state of an interpreter at program point $*$ in Figure 1.

[1] The `null` constant is decorated with the class κ induced by its context, as in $v := \texttt{null}\ \kappa$, where κ is the type of v. This way we do not need a distinguished type for `null`. We assume that the compiler provides κ.

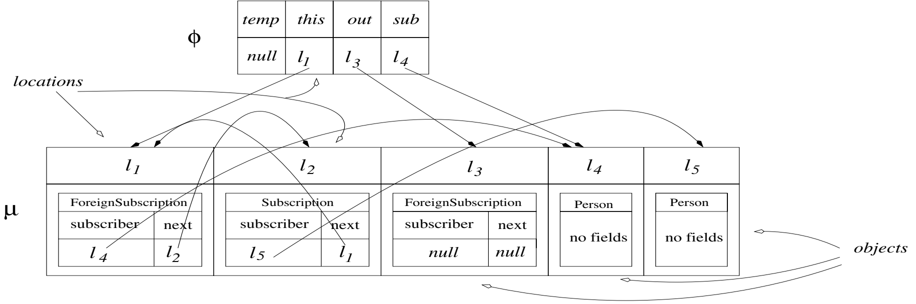


Fig. 3. A state (frame ϕ and memory μ) for the type environment $\tau = [temp \mapsto \texttt{Subs}, this \mapsto \texttt{ForeignSubs}, out \mapsto \texttt{ForeignSubs}, sub \mapsto \texttt{Person}]$

The states of the computation are made of a frame ϕ and a memory μ. We assume type correctness $\phi \star \mu : \tau$, which bans dangling pointers and insists that variables and fields are bound to *null* or to locations containing objects allowed by their type. This constraint is sensible for strongly-typed languages which use a conservative garbage collector. For a formal definition, see [7].

Definition 4. *Let τ be the type environment at a given program point p. The set of possible* states *at p is $\Sigma_\tau = \{\phi \star \mu \mid \phi \in Frame_\tau,\ \mu \in Memory,\ \phi \star \mu : \tau\}$.*

Example 4. In Figure 3, the pair $\phi \star \mu$ does not contain dangling pointers and respects the typing of variables and fields. Hence $\phi \star \mu : \tau$ and $\phi \star \mu \in \Sigma_\tau$.

Denotations are the input/output semantics of a piece of code. *Interpretations* provide a denotation for each method.

Definition 5. *An* interpretation *I maps methods to* denotations *such that, for each method $\kappa.\mathtt{m}$, we have $I(\kappa.\mathtt{m}) : \Sigma_{input(\kappa.\mathtt{m})} \to \Sigma_{output(\kappa.\mathtt{m})}$.*

We describe here informally the denotations for our language. Their formal definition is in [7].

Expressions have side-effects and return a value. Hence their denotations are partial maps from an initial to a final state containing a distinguished variable *res* holding the expression's value: $\mathcal{E}^I_\tau[\![_]\!] : exp \mapsto (\Sigma_\tau \to \Sigma_{\tau+exp})$, where $res \notin \mathrm{dom}(\tau)$, $\tau + exp = \tau[res \mapsto type_\tau(exp)]$ and I is an interpretation. Namely, given an input state $\phi \star \mu$, the denotation of $\mathtt{null}\ \kappa$ binds *res* to *null* in ϕ. That of $\mathtt{new}\ \kappa$ binds *res* to a new location holding a new object of class κ. That of v copies v into *res*. That of $v.\mathtt{f}$ accesses the object $o = \mu(\phi(v))$ bound to v (provided $\phi(v) \neq null$) and copies the field $\mathtt{f}$ of o (*i.e.,* $o.\phi(\mathtt{f})$) into *res*. That of $(\kappa)v$ copies v into *res*, but only if the cast is satisfied. That of method call uses the dynamic class of the receiver to fetch the denotation of the method from I. It plugs it in the calling context, by building a starting state $\sigma^\dagger$ for the method, whose formal parameters (including *this*) are bound to the actual parameters.

The denotation of a command is a partial map from an initial to a final state: $\mathcal{C}^I_\tau[\![_]\!] : com \mapsto (\Sigma_\tau \to \Sigma_\tau)$ with $res \notin \mathrm{dom}(\tau)$. The denotation of v:=exp uses

that of *exp* to get a state where *res* holds *exp*'s value. Then it copies *res* into v and removes *res*. Similarly for $v.\mathtt{f} := exp$, but *res* is copied into the field `f` of the object bound to v, if any. The denotation of the conditionals checks their guard and then uses the denotation of `then` or that of `else`. The denotation of a sequence of commands is the functional composition of their denotations.

We use $\mathcal{C}^I_\tau[\![_]\!]$ to define a transformer on interpretations. It evaluates the methods' bodies in I, expanding the input state with local variables bound to *null*. It restricts the final state to *out*, to respect Definition 5. This corresponds to the *immediate consequence operator* used in logic programming [3]. Its least fixpoint is the *denotational semantics* of the program (see [7]).

5 Non-cyclicity

A variable v is non-cyclical if there is no cycle in its *reachable* locations. Namely, if a location l is bound, in a memory μ, to an object $o = \mu(l)$, then the locations bound to o's fields (*i.e.,* $\mathrm{rng}(o.\phi) \cap Loc$) are reachable from l. Reachability is the transitive closure of this relation, passing through one or more objects.

Definition 6. *Let $\mu \in Memory$ and $l \in \mathrm{dom}(\mu)$. The set of locations* reachable *in μ from l is $L(\mu)(l) = \cup\{L^i(\mu)(l) \mid i \geq 0\}$, where $L^i : Memory \mapsto Loc \mapsto \wp(Loc)$, for $i \geq 0$, is such that $L^0(\mu)(l) = \mathrm{rng}(\mu(l).\phi) \cap Loc$ and $L^{i+1}(\mu)(l) = \cup\{\mathrm{rng}(\mu(j).\phi) \cap Loc \mid j \in L^i(\mu)(l)\}$.*

We let *all* fields $\mathrm{rng}(\mu(j).\phi)$ of the object $\mu(j)$ to be reachable, since we allow checked casts (Section 4). Then all fields of an object can be accessed.

Example 5. In Figure 3 we have $L^0(\mu)(l_1) = \{l_2, l_4\}$, $L^1(\mu)(l_1) = \{l_1, l_5\}$ and $L^2(\mu)(l_1) = \{l_2, l_4\} = L^0(\mu)(l_1)$; $L^0(\mu)(l_3) = L^0(\mu)(l_4) = \varnothing$. Hence $L(\mu)(l_1) = \{l_1, l_2, l_4, l_5\}$ and $L(\mu)(l_3) = L(\mu)(l_4) = \varnothing$. Hence l_1 is reachable from l_1, while no location is reachable from l_3 nor from l_4.

A location l is not necessarily reachable from itself, as Example 5 shows for l_3 and l_4. That is only true if there is a path starting at l and passing back through l. This is the case of l_1 in Figure 3, so that we say that l_1 is *cyclical* there. But l is cyclical also if a location l' can be reached from l and l' is cyclical.

Definition 7. *Let $\mu \in Memory$. A location $l \in \mathrm{dom}(\mu)$ is* cyclical *in μ if there is $l' \in L(\mu)(l)$ such that $l' \in L(\mu)(l')$. A variable $v \in \mathrm{dom}(\tau)$ is* cyclical *in $\phi \star \mu \in \Sigma_\tau$ if $\phi(v) \neq null$ and $\phi(v)$ is cyclical in μ. It is* non-cyclical *otherwise.*

We can hence define the set of states where all variables in a set *nc* are non-cyclical. Nothing is known about the others.

Definition 8. *Let $nc \subseteq \mathrm{dom}(\tau)$. The set of states in Σ_τ where the variables nc are non-cyclical is $\gamma_\tau(nc) = \{\phi \star \mu \in \Sigma_\tau \mid nc \text{ are non-cyclical in } \phi \star \mu\}$.*

Example 6. Let $\phi \star \mu$ be as in Figure 3. By Example 5, $\phi \star \mu \in \gamma_\tau(\{temp, out, sub\})$, $\phi \star \mu \in \gamma_\tau(\{temp, out\})$, $\phi \star \mu \in \gamma_\tau(\{sub\})$, $\phi \star \mu \in \gamma_\tau(\varnothing)$ but $\phi \star \mu \notin \gamma_\tau(\{this, out\})$, since variable *this* is cyclical in $\phi \star \mu$.

Definition 8 might suggest that an abstract domain for definite non-cyclicity is $\wp(\mathrm{dom}(\tau))$. However, this would result in a Galois *connection* rather than a Galois *insertion* with $\wp(\Sigma_\tau)$. For instance, in Figure 3 we have $\tau(sub) = \mathtt{Person}$. It is hence redundant to ask *sub* to be non-cyclical: no cycle can be reached from a `Person`. As a consequence, $\gamma_\tau(\{sub\}) = \gamma_\tau(\varnothing)$ *i.e.,* γ_τ is not one-to-one and, hence, is not the concretisation map of a Galois *insertion* (Section 3).

To get a Galois insertion, we must consider only those variables whose static type is *cyclical* so that they can be both cyclical and non-cyclical at run-time. To define cyclicity for types (classes), we first need a notion of reachability for types (classes). From a class κ, we can reach the classes $C = \mathrm{rng}(F(\kappa))$ of its fields $F(\kappa)$ and those of every subclass $\kappa' \leq \kappa$. Moreover, we can reach every subclass of C (because of the cast, see Section 4). We can also reach the classes of the fields of C, and so on recursively.

Definition 9. *The classes reachable from* $\kappa \in \mathcal{K}$ *are* $C(\kappa) = \cup\{C^i(\kappa) \mid i \geq 0\}$ *where* $C^0(\kappa) = \downarrow\{\mathrm{rng}(F(\kappa')) \mid \kappa' \leq \kappa\}$ *and* $C^{i+1}(\kappa) = \downarrow\{\mathrm{rng}(F(\kappa')) \mid \kappa' \in C^i(\kappa)\}$.

Example 7. In Figure 1, for every $i \geq 0$ we have that $C^i(\mathtt{Person}) = \varnothing$ and $C^i(\mathtt{Object}) = C^i(\mathtt{Subs}) = C^i(\mathtt{ForeignSubs}) = \{\mathtt{Person}, \mathtt{Subs}, \mathtt{ForeignSubs}\}$. Thus $C(\mathtt{Person}) = \varnothing$ and $C(\mathtt{Object}) = C(\mathtt{Subs}) = C(\mathtt{ForeignSubs}) = \{\mathtt{Person}, \mathtt{Subs}, \mathtt{ForeignSubs}\}$.

We can now define cyclicity for classes.

Definition 10. *Let* $\kappa \in \mathcal{K}$*. Class* κ *is cyclical if and only if there exists* $\kappa' \in C(\kappa)$ *such that* $\kappa' \in C(\kappa')$*. Given* τ*, the set of variables of non-cyclical static type is* $NC_\tau = \{v \in \mathrm{dom}(\tau) \mid \tau(v) \text{ is non-cyclical}\}$.

Example 8. From Example 7 we conclude that `Object`, `Subs` and `ForeignSubs` in Figure 1 are cyclical, while `Person` is non-cyclical. `Object` is recognised as cyclical since there is $\mathtt{Subs} \in C(\mathtt{Object})$ and $\mathtt{Subs} \in C(\mathtt{Subs})$. Similarly, `ForeignSubs` is recognised as cyclical by taking $\kappa' = \mathtt{ForeignSubs}$.

We can now define the abstract domain for definite non-cyclicity. Its elements are those subsets of $\wp(\mathrm{dom}(\tau))$ which contain all variables of non-cyclical type.

Definition 11. *The abstract domain for definite non-cyclicity is* $\mathsf{NC}_\tau = \{nc \in \wp(\mathrm{dom}(\tau)) \mid NC_\tau \subseteq nc\}$*, ordered by inverse set-inclusion (*$\mathrm{dom}(\tau)$ *is the least element,* NC_τ *is the top element). Its concretisation map is* γ_τ *in Definition 8, restricted to* NC_τ*.*

Example 9. From Example 8, in Figure 3 $NC_\tau = \{sub\}$ and $\{temp, out, sub\} \in \mathsf{NC}_\tau$, $\{this, out, sub\} \in \mathsf{NC}_\tau$, $\{sub\} \in \mathsf{NC}_\tau$ but $\varnothing \notin \mathsf{NC}_\tau$ and $\{this\} \notin \mathsf{NC}_\tau$.

We state now the main result of this section *i.e.,* our abstract domain for non-cyclicity actually induces a Galois insertion with the concrete domain.

Proposition 1. *The map* γ_τ *of Definition 8 is the concretisation map of a Galois insertion between* NC_τ *and* $\wp(\Sigma_\tau)$*. The induced abstraction map, for any* $S \subseteq \Sigma_\tau$*, is* $\alpha_\tau(S) = \{v \in \mathrm{dom}(\tau) \mid v \text{ is non-cyclical in every } \phi \star \mu \in S\}$.

6 Abstract Semantics for Non-cyclicity

We define here an abstract denotational semantics over NC. It builds a chain of *non-cyclicity interpretations* until a fixpoint is reached [3].

Definition 12. *A* non-cyclicity interpretation *I maps methods to* abstract denotations*, such that* $I(\kappa.\mathtt{m}) : \mathsf{NC}_{input(\kappa.\mathtt{m})} \mapsto \mathsf{NC}_{output(\kappa.\mathtt{m})}$ *for each method* $\kappa.\mathtt{m}$*.*

Example 10. A non-ciclicity interpretation for Figure 1 might be such that $I(\mathtt{Subs.foreign})(nc) = I(\mathtt{ForeignSubs.foreign})(nc) = \{out\}$ for any nc. That is, the output of such methods is non-cyclical, whatever is the input. This is sensible since they return a new, non-cyclical `ForeignSubs`, if they do not diverge. Interpretation I is the *bottom* interpretation which, for every method $\kappa.\mathtt{m}$, maps every input to the least element of $\mathsf{NC}_{output(\kappa.\mathtt{m})}$.

Proposition 2. *Figures 4 and 5 report abstract denotations for our language. They are correct w.r.t. the concrete denotations described in Section 4.*

Note that Figures 4 and 5 use preliminary information on the pairs of variables which *share i.e.,* are bound to overlapping data-structures [8].

We have used abstract interpretation to derive Figures 4 and 5. Namely, for every concrete operation (*denotation*, since we deal with denotational semantics) op and abstract input nc, those figures provide a correct approximation nc' of $\alpha(op(\gamma(nc)))$. This means that the variables in nc' are definitely non-cyclical in every concrete state σ' obtained by applying op to any concrete state σ where the variables in nc are definitely non-cyclical. Let us discuss those figures.

For expressions, we have $\mathcal{NCE}^I_\tau[\![exp]\!] : \mathsf{NC}_\tau \mapsto \mathsf{NC}_{\tau+exp}$, where $res \notin \mathrm{dom}(\tau)$ and I is a non-cyclicity interpretation. Variable res, in the final state, refers to the value of exp (Section 4). Hence, on NC, variable res belongs to nc' if the analysis is able to prove that the value of exp is definitely non-cyclical.

$$\mathcal{NCE}^I_\tau[\![\mathtt{null}\ \kappa]\!](nc) = \mathcal{NCE}^I_\tau[\![\mathtt{new}\ \kappa]\!](nc) = nc \cup \{res\}$$

$$\mathcal{NCE}^I_\tau[\![v]\!](nc) = \mathcal{NCE}^I_\tau[\![(\kappa)v]\!](nc) = \begin{cases} nc \cup \{res\} & \text{if } v \in nc \\ nc & \text{otherwise} \end{cases}$$

$$\mathcal{NCE}^I_\tau[\![v.\mathtt{f}]\!](nc) = \begin{cases} nc \cup \{res\} & \text{if } v \in nc \text{ or } F(\tau(v))(\mathtt{f}) \text{ is non-cyclical} \\ nc & \text{otherwise} \end{cases}$$

$$\mathcal{NCE}^I_\tau[\![v.\mathtt{m}(v_1, \ldots v_n)]\!](nc) = \begin{cases} (nc \setminus SE) \cup \{res\} & \text{if } nc'' = \{out\} \\ nc \setminus SE & \text{if } nc'' = \varnothing \end{cases}$$

$$\text{where } nc^\dagger = (nc \cap \{v, v_1, \ldots, v_n\})[v \mapsto \mathit{this}, v_1 \mapsto w_1, \ldots, v_n \mapsto w_n]$$

$$nc'' = \cap\{I(\kappa.\mathtt{m})(nc^\dagger) \mid \kappa.\mathtt{m} \text{ can be called here}\}$$

$$\text{and } SE = \{w \in \mathrm{dom}(\tau) \mid w \text{ shares with some } \{v, v_1, \ldots, v_n\}\} \setminus NC_\tau.$$

Fig. 4. The abstract denotational semantics of the expressions

$$\mathcal{NCC}^I_\tau[\![v{:=}exp]\!](nc) = \mathcal{NCE}^I_\tau[\![exp]\!](nc) \circ setVar^v_{\tau+exp}$$

$$\text{where } setVar^v_{\tau'}(nc) = \begin{cases} (nc \setminus \{res\}) \cup \{v\} & \text{if } res \in nc \\ nc \setminus \{v\} & \text{otherwise} \end{cases}$$

$$\mathcal{NCC}^I_\tau[\![v.\mathtt{f} := exp]\!](nc) = \mathcal{NCE}^I_\tau[\![exp]\!](nc) \circ setField^{v.\mathtt{f}}_{\tau+exp}$$

$$\text{where } setField^{v.\mathtt{f}}_{\tau'}(nc) = \begin{cases} nc \setminus \{res\} & \text{if } F(\tau'(v))(\mathtt{f}) \text{ is non-cyclical} \\ & \text{or } (res \in nc \text{ and } res \text{ does not share with } v) \\ (nc \setminus SE) \setminus \{res\} & \text{otherwise} \end{cases}$$

$$\text{where } SE = \{w \in \mathrm{dom}(\tau') \mid w \text{ shares with } v\} \setminus NC_{\tau'}$$

$$\mathcal{NCC}^I_\tau\left[\!\!\left[\begin{array}{l}\mathtt{if}\ v = w\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right](nc) = \begin{cases} \mathcal{NCC}^I_\tau[\![com_1]\!](nc \cup \{v, w\}) \cap \mathcal{NCC}^I_\tau[\![com_2]\!](nc) \\ \quad \text{if } v \in nc \text{ or } w \in nc \\ \mathcal{NCC}^I_\tau[\![com_1]\!](nc) \cap \mathcal{NCC}^I_\tau[\![com_2]\!](nc) \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{NCC}^I_\tau\left[\!\!\left[\begin{array}{l}\mathtt{if}\ v =\ \mathtt{null}\ \mathtt{then}\ com_1 \\ \mathtt{else}\ com_2\end{array}\right]\!\!\right](nc) = \mathcal{NCC}^I_\tau[\![com_1]\!](nc \cup \{v\}) \cap \mathcal{NCC}^I_\tau[\![com_2]\!](nc)$$

$$\mathcal{NCC}^I_\tau[\![\{\}]\!] = \lambda nc \in \mathsf{NC}_\tau.nc, \quad \mathcal{NCC}^I_\tau[\![\{com_1; \ldots; com_p\}]\!] = \mathcal{NCC}^I_\tau[\![com_1]\!] \circ \cdots \circ \mathcal{NCC}^I_\tau[\![com_p]\!].$$

Fig. 5. The abstract denotational semantics of the commands

The concrete denotations of the expressions `null` κ and `new` κ yield a final state σ' which coincides with σ except on *res*, which holds *null* or a new object, respectively. In both cases, *res* is non-cyclical and we define $nc' = nc \cup \{res\}$.

The concrete denotation of v computes σ' by copying the value of v into *res*, while the other variables are not affected. Hence *res* belongs to nc' if and only if $v \in nc$. The same is correct for the cast $(\kappa)v$, whose concrete denotation coincides with that of v when it is defined (*i.e.,* when the cast is legal).

The concrete denotation of $v.\mathtt{f}$ loads in *res* the value of the field `f` of the object bound to v, if any. The other variables are not changed. Then if v is non-cyclical in σ also *res* is non-cyclical in σ', since you can reach *res* from a field of v. If, instead, we do not know if v is non-cyclical in σ (*i.e.,* if $v \notin nc$), we can still guarantee that *res* is non-cyclical in σ' if we know that the class $F(\tau(v))(\mathtt{f})$ of the field `f` is non-cyclical. In conclusion, we let $nc' = nc \cup \{res\}$ if $v \in nc$ or $F(\tau(v))(\mathtt{f})$ is non-cyclical, while we conservatively assume $nc' = nc$ otherwise.

The concrete denotation of the method call $v.\mathtt{m}(v_1, \ldots, v_n)$ first computes an input state $\sigma^\dagger$ for the method. It consists of the input state σ restricted to $v, v_1, \ldots, v_n$ and where v is renamed into *this* and each v_i into the formal parameter w_i. We mimic this behaviour on the abstract domain and define $nc^\dagger$ accordingly. We then apply $\sigma^\dagger$ to the interpretation for the method. In the concrete semantics, late-binding is resolved by using the run-time class κ of v. In the abstract semantics, we only know that $\kappa \leq \tau(v)$. Hence we conservatively select *all* possible targets $\kappa.\mathtt{m}$ of the method call. The final state of the call is chosen to be consistent with all such targets, by using set intersection: $nc'' = \cap\{I(\kappa.\mathtt{m})(nc^\dagger) \mid \kappa.\mathtt{m} \text{ can be called here}\}$. If *out* is non-cyclical in nc'' then the value of the method call expression is non-cyclical also, and we should define

$nc' = nc \cup \{res\}$, otherwise we should let $nc' = nc$. But you can see in Figure 4 that we remove from nc a set of variables SE which share. with at least one of $v, v_1, \ldots, v_n$, and have cyclical type. This is because a method can modify, as a *side effect*, everything which is reachable from its formal parameters, and hence introduce cyclicity. Without sharing information, we could only conservatively assume that every pair of variables shares, when their types allow them to share. This definition can also be made more precise by including *shadow copies* of $v, v_1, \ldots, v_n$ in the method body. They hold the initial values of such parameters and are never modified, so that at the end of the method call they provide explicit information on the cyclicity of $v, v_1, \ldots, v_n$.

For commands, we have $\mathcal{NCC}^I_\tau[\![com]\!] : \mathsf{NC}_\tau \mapsto \mathsf{NC}_\tau$ where I is a non-ciclicity interpretation and $res \notin \mathrm{dom}(\tau)$.

The concrete denotation of v:=exp first evaluates exp and then composes its denotation with a map $setVar^v$ which copies res into v. The initial value of v is lost. This is mimicked on NC by an abstract map $setVar^v$. If variable res is non-cyclical, this map makes v non-cyclical also. Otherwise it removes v from the set of non-cyclical variables, since we have no information on its cyclicity.

The concrete denotation of $v.\mathtt{f}$:=exp uses, similarly, a map *setField* which updates σ by writing the value of exp, held in res, in the field $\mathtt{f}$ of v, thus yielding σ'. Hence, if res is non-cyclical and does not share with v, this operation can only remove cyclicity and we can safely assume $nc' = nc \setminus \{res\}$ (variable res is removed after the assignment). Similarly when the field $\mathtt{f}$ has non-cyclical type, so that we cannot reach a cycle from $\mathtt{f}$. The non-sharing requirement is necessary since otherwise v might be made cyclical by closing a loop, like in $v.\mathtt{f}$:= v. If none of these cases applies, we might make cyclical the variables SE which share with v and have cyclical type (often $v \in SE$).

The concrete denotation of the conditionals can be conservatively approximated by the greatest lower bound (*i.e.,* set intersection, see Definition 11) of their two branches. We improve this approximation by taking the guard into account. If $v = w$ holds then v and w are aliases and hence both cyclical or both non-cyclical. If $v = \mathtt{null}$ holds then v contains *null* and is hence non-cyclical.

The concrete denotation of the sequential composition of commands is approximated by the functional composition of their abstract denotations.

Definition 13. *The* transformer on interpretations *transforms a non-cyclicity interpretation I in a new non-cyclicity interpretation I' such that* $I'(\kappa.\mathtt{m})(nc) = \mathcal{NCC}^I_{scope(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!](nc \cup \{out, w_{n+1}, \ldots, w_{n+m}\}) \cap \{out\}$. *The* denotational non-ciclicity semantics *of a program is the least fixpoint of this transformer.*

In Definition 13, local variables $\{w_{n+1}, \ldots, w_{n+m}\}$ are assumed to be non-cyclical at the beginning of the methods, since they are bound to *null* there.

Proposition 3. *The non-cyclicity semantics of Definition 13 is correct w.r.t. the concrete semantics of Section 4.*

The least fixpoint of Definition 13 is computed by repeated application of this transformer from the bottom interpretation [5, 3]. In our case, the bottom interpretation is given in Example 10. Example 11 shows that one application of the transformer reaches the fixpoint.

Example 11. Let us prove that the abstract interpretation I of Example 10 is a fixpoint of the transformer of Definition 13: $\mathcal{NCC}^I_{input(\kappa.\mathtt{m})}[\![body(\kappa.\mathtt{m})]\!](nc \cup \{out, temp\}) = \{out\}$ for any $nc \in \mathsf{NC}_{input(\kappa.\mathtt{m})}$, where $\kappa.\mathtt{m}$ ranges over `Subs.foreign` and `ForeignSubs.foreign` (methods `monthlyCost` are irrelevant since they work on primitive types). We can only have $nc = \varnothing$ or $nc = \{this\}$. By monotonicity of the denotations in Figures 4 and 5, we can just prove this result for $nc = \varnothing > \{this\}$ *i.e.,* assuming that we do not know anything about the non-ciclicity of the variable *this*.

$$
\begin{array}{c}
\{out, sub, temp\}\\
sub := this.\mathtt{subscriber}\\
\{out, sub, temp\}\\
out := \mathtt{new\ ForeignSubs}\\
\{out, sub, temp\}\\
temp := this.\mathtt{next}\\
\{out, sub\}\\
\mathtt{if}\ (temp = \mathtt{null})\ \mathtt{then}\ \{\}\\
\mathtt{else}\ out.\mathtt{next} := temp.\mathtt{foreigners}()\\
\{out, sub\}\\
out.\mathtt{subscriber} := sub\\
\{out, sub\}
\end{array}
$$

Fig. 6. The analysis of method `ForeignSubs.foreign`

Consider `Subs.foreign`. Let $\tau = scope(\mathtt{Subs.foreign}) = [this \mapsto \mathtt{Subs}, temp \mapsto \mathtt{Subs}, out \mapsto \mathtt{ForeignSubs}]$. Then $\mathcal{NCC}^I_\tau[\![temp := this.\mathtt{next}]\!](\{out, temp\})$ is equal to $setVar^{temp}_{\tau+this.\mathtt{next}}(\mathcal{NCE}^I_\tau[\![this.\mathtt{next}]\!](\{out, temp\}))$, which is $setVar^{temp}_{\tau+this.\mathtt{next}}(\{out, temp\}) = \{out\}$. Hence

$$
\begin{aligned}
&\mathcal{NCC}^I_\tau\left[\!\!\left[\begin{array}{l} temp := this.\mathtt{next};\\ \mathtt{if}\ temp = \mathtt{null\ then}\ \{\}\ \mathtt{else}\ out := temp.\mathtt{foreign}()\end{array}\right]\!\!\right]\left(\left\{\begin{array}{c} out,\\ temp\end{array}\right\}\right)\\
&=\mathcal{NCC}^I_\tau\left[\!\!\left[\begin{array}{l}\mathtt{if}\ temp = \mathtt{null\ then}\ \{\}\\ \mathtt{else}\ out := temp.\mathtt{foreign}()\end{array}\right]\!\!\right](\mathcal{NCC}^I_\tau[\![temp := this.\mathtt{next}]\!](\{out, temp\}))\\
&= \mathcal{NCC}^I_\tau[\![\mathtt{if}\ temp = \mathtt{null\ then}\ \{\}\ \mathtt{else}\ out := temp.\mathtt{foreign}()]\!](\{out\})\\
&= \mathcal{NCC}^I_\tau[\![\{\}]\!](\{temp, out\}) \cap \mathcal{NCC}^I_\tau[\![out := temp.\mathtt{foreign}()]\!](\{out\})\\
&= \{temp, out\} \cap setVar^{out}_{\tau+temp.\mathtt{foreign}()}(\mathcal{NCE}^I_\tau[\![temp.\mathtt{foreign}()]\!](\{out\})).
\end{aligned}
$$

In this call, $nc^\dagger = \varnothing$ and $\tau(temp) = \mathtt{Subs}$. Then $nc'' = I(\mathtt{Subs.foreign})(\varnothing) \cap I(\mathtt{ForeignSubs.foreign})(\varnothing) = \{out\} \cap \{out\} = \{out\}$. A sharing analysis, such as that in [8], proves that *temp* shares with only *this* and *temp* itself here (*out* holds *null*). So $SE = \{this, temp\}$ and $\mathcal{NCE}^I_\tau[\![temp.\mathtt{foreign}()]\!](\{out\}) = (\{out\} \setminus SE) \cup \{res\} = \{out, res\}$ and the equation above is $\{temp, out\} \cap \big(setVar^{out}_{\tau+temp.\mathtt{foreign}()}(\{out, res\})\big) = \{temp, out\} \cap \{out\} = \{out\}$.

The result for `ForeignSubs.foreign` is shown in Figure 6. We report the approximation before and after each statement. We do not consider the statement $out.\mathtt{numOfChannels} := this.\mathtt{numOfChannels}$ in Figure 1 since it deals with primitive types, not relevant for us. The result is $\{out, sub\}$ *i.e., out* and *sub* are non-cyclical. Its restriction to *out* (Definition 13) is $\{out\}$, as expected. The assignment $out.\mathtt{next} := temp.\mathtt{foreign}()$ maintains *out*'s non-cyclicity since the value of the right-hand side is non-cyclical (as we have computed above) and does

not share with *out*, as it can be proved for instance through the analysis in [8]. So the first case of the definition of *setField* in Figure 5 applies. The assignment *out*.`subscriber` := *sub* maintains *out*'s non-cyclicity since field `subscriber` has non-cyclical type `Person`. The first case of the definition of *setField* applies.

Example 11 shows that our analysis is able to prove non-cyclicity in a non-trivial situation. Example 12 shows instead that, correctly, non-cyclicity is lost when a variable is bound to a cyclic data-structure.

Example 12. Let only v be in scope. At the end of the piece of code v:=`new Subs`; v.`next` :=v, variable v is cyclical. Our analysis reflects this since, if we start it for instance from $\varnothing$ (no variable is definitely non-cyclical) then the approximation $\{v\}$ is computed after the first statement, and $\varnothing$ after the second. Here, we used the rule for v.`f` :=*exp* in Figure 5 with $SE = \{res, v\}$, since *res i.e.,* the value of v, shares with v.

We conclude with Example 13, which shows a *false alarm i.e.,* a situation where our analysis is too conservative and is not able to prove non-cyclicity.

Example 13. Let only v be in scope. At the end of the piece of code v:=`new Subs`; v.`next` := `new Subs`; v.`next` :=v.`next`, variable v is non-cyclical. If we start for instance our analysis from $\varnothing$, we compute $\{v\}$ after the first and second statement. For the last one we apply the rule for v.`f` :=*exp* in Figure 5 with $SE = \{res, v\}$, since *res i.e.,* the value of v.`next`, shares with v. The result, as in Example 12, is $\varnothing$ *i.e.,* the analysis is too conservative to prove that v is non-cyclical.

7 Compilation into Boolean Functions

Figures 4 and 5 report maps over NC *i.e.,* over sets of variables. They can be represented through Boolean functions (or formulas) *i.e.,* functions over Boolean variables, which can then be efficiently implemented through *binary decision diagrams* [4]. The idea is that two Boolean variables $\check{v}$ and $\hat{v}$ represent the non-cyclicity of program variable v in the input, respectively, output, of a denotation.

Definition 14. *A non-cylicity denotation* $d : \mathsf{NC}_\tau \mapsto \mathsf{NC}_\tau$ *is* represented *by a Boolean function* ϕ *over the variables* $\{\check{v} \mid v \in \mathrm{dom}(\tau)\} \cup \{\hat{v} \mid v \in \mathrm{dom}(\tau')\}$ *iff for every* $nc \in \mathsf{NC}_\tau$ *we have* $d(nc) = nc' \Leftrightarrow \{v \mid (\wedge\{\check{v} \mid v \in nc\} \wedge \phi) \models \hat{v}\} = nc'$, *where* $\models$ *is logical consequence or entailment.*

Example 14. Let τ be as in Figure 3. The denotation $d : \mathsf{NC}_\tau \mapsto \mathsf{NC}_\tau$ such that $d(\{this, sub\}) = \{out, sub\}$ and $d(nc) = \{sub\}$ for any $nc \neq \{this, sub\}$, is represented by $\phi = \big((\check{this} \wedge \neg\check{out} \wedge \neg\check{temp} \wedge \check{sub}) \rightarrow \hat{out}\big) \wedge \hat{sub}$.

Composition $d_1 \circ d_2$ of non-cyclicity denotations d_1 and d_2 is represented by the Boolean function $b_1 \circ b_2 = \exists_{\mathbf{v}} \ (b_1[\hat{\mathbf{v}} \mapsto \mathbf{v}'] \wedge b_2[\check{\mathbf{v}} \mapsto \mathbf{v}'])$, where b_1 is the Boolean representation of d_1, b_2 is the representation of d_2, $b_1[\hat{\mathbf{v}} \mapsto \mathbf{v}']$ renames the output variables of b_1 into new temporary, primed variables, $b_2[\check{\mathbf{v}} \mapsto \mathbf{v}']$ renames the

$$\mathcal{BE}_\tau[\![\texttt{null}\ \kappa]\!] = \mathcal{BE}_\tau[\![\texttt{new}\ \kappa]\!] = \hat{res} \wedge U(\mathrm{dom}(\tau))$$

$$\mathcal{BE}_\tau[\![v]\!] = \mathcal{BE}_\tau[\![(\kappa)v]\!] = (\check{v} \to \hat{res}) \wedge U(\mathrm{dom}(\tau))$$

$$\mathcal{BE}_\tau[\![v.\texttt{f}]\!] = \begin{cases} \hat{res} \wedge U(\mathrm{dom}(\tau)) & \text{if } F(\tau(v))(\texttt{f}) \text{ is non-cyclical} \\ (\check{v} \to \hat{res}) \wedge U(\mathrm{dom}(\tau)) & \text{otherwise} \end{cases}$$

$$\mathcal{BE}_\tau[\![v.\texttt{m}(v_1,\ldots,v_n)]\!] = U(\mathrm{dom}(\tau) \setminus SE) \wedge [(\check{v} \to \hat{this} \wedge \check{v}_1 \to \hat{w}_1 \wedge \ldots \wedge \check{v}_n \to \hat{w}_n) \circ$$
$$\circ \vee\{I(\kappa.\texttt{m}) \mid \kappa.\texttt{m} \text{ can be called here}\} \circ (\check{out} \to \hat{res})]$$

$$\mathcal{BC}_\tau[\![v{:=}exp]\!] = \mathcal{BE}_\tau[\![exp]\!] \circ setVar^v_{\tau+exp}$$

$$\text{with } setVar^v_{\tau'} = (\check{res} \to \hat{v}) \wedge U(\mathrm{dom}(\tau') \setminus \{res, v\})$$

$$\mathcal{BC}_\tau[\![v.\texttt{f} := exp]\!] = \mathcal{BE}_\tau[\![exp]\!] \circ setField^{v.\texttt{f}}_{\tau+exp}$$

$$\text{with } setField^{v.\texttt{f}}_{\tau'} = \begin{cases} U((\mathrm{dom}(\tau')\setminus SE)\setminus\{res\}) & \text{if } res \text{ and } v \text{ share and } F(\tau(v))(\texttt{f}) \text{ is cycl.} \\ (\wedge\{(\check{res} \wedge \check{v}) \to \hat{v} \mid v \in SE\}) \wedge U((\mathrm{dom}(\tau') \setminus SE) \setminus \{res\}) & \text{else} \end{cases}$$

$$\mathcal{BC}_\tau[\![\texttt{if}\ v = w\ \texttt{then}\ com_1\ \texttt{else}\ com_2]\!] = ((\check{v} \leftrightarrow \check{w}) \wedge \mathcal{BC}_\tau[\![com_1]\!]) \vee \mathcal{BC}_\tau[\![com_2]\!]$$

$$\mathcal{BC}_\tau[\![\texttt{if}\ v = \texttt{null}\ \texttt{then}\ com_1\ \texttt{else}\ com_2]\!] = (\check{v} \wedge \mathcal{BC}_\tau[\![com_1]\!]) \vee \mathcal{BC}_\tau[\![com_2]\!]$$

$$\mathcal{BC}_\tau[\![\{\}]\!] = U(\mathrm{dom}(\tau)), \quad \mathcal{BC}_\tau[\![\{com_1; \ldots; com_p\}]\!] = \mathcal{BC}_\tau[\![com_1]\!] \circ \cdots \circ \mathcal{BC}_\tau[\![com_p]\!].$$

Fig. 7. Compilation rules from our language into Boolean functions

input variables of b_2 into the same temporaries and $\exists_{\mathbf{v}}$ removes such temporaries through Schröder elimination [2]: $\exists_x \phi = \phi[x \mapsto true] \vee \phi[x \mapsto false]$.

Figure 7 reports Boolean functions for the non-cyclicity denotations of Figures 4 and 5. The *frame condition* $U(vars) = \wedge\{\check{v} \to \hat{v} \mid v \in vars\}$ states that variables *vars* do not change. Interpretations I map now methods to Boolean functions representing their non-cyclicity denotation. You can see Figure 7 as a set of *compilation rules* from the language of Section 4 into Boolean functions.

Let us consider Figure 7. The representation for `new` κ and `null` κ is a Boolean function stating that *res* is non-cyclical in the output. All other variables are unchanged. That of v and $(\kappa)v$ propagates the non-cyclicity of v into that of *res*. The representation of $v.\texttt{f}$ depends on the non-cyclicity of $F(\tau(v))(\texttt{f})$, which can be checked at analysis-time. The representation for method call is the composition of a Boolean function, matching the actual parameters with the formal ones, with a Boolean function that fetches the interpretations of the methods which might be called (I is fed later), and a Boolean function which renames *out* into *res*. A frame condition expresses that no variable is changed by the call, except those in *SE* (Figure 4). Assignments use, as in Figure 5, maps (now formulas) *setVar* and *setField*. The latter checks, at analysis-time, if *res* shares with v and field `f` has cyclical type. The representation of the conditionals is the disjunction of their two branches, but we improve the information available on the `then` branch, exactly as in Figure 5. Namely, if $v = w$ holds then v and w are both cyclical or both non-cyclical. If $v =$ `null` holds then v is non-cyclical.

Example 15. Figure 2 shows the application of the compilation rules in Figure 7 to the abstract compilation of the method `ForeignSubs.foreign` in Figure 1.

Interpretations over Boolean functions are updated by a transformer which is the compilation into Boolean functions of that of Definition 13.

Definition 15. *The* transformer on non-cyclicity interpretations *transforms a non-cyclicity interpretation I' into I'' (both represented with Boolean functions) such that* $I''(\kappa.\mathtt{m}) = (U(\mathrm{dom}(input(\kappa.\mathtt{m}))) \wedge \hat{out} \wedge \hat{w}_{n+1} \wedge \ldots \wedge \hat{w}_{n+m}) \circ \phi_{\kappa.\mathtt{m}}[I \mapsto I'] \circ (\check{out} \to \hat{out})$*, where* $\phi_{\kappa.\mathtt{m}}[I \mapsto I']$ *is the compiled body (formula) of method* $\kappa.\mathtt{m}$ *with I' plugged instead of I. The* denotational non-cyclicity semantics over Boolean functions *of a program is the least fixpoint of this transformer.*

Proposition 4. *The denotational non-cyclicity semantics over Boolean functions represents (Definition 14) the non-cyclicity semantics of Definition 13.*

Example 16. The bottom interpretation of Example 10 is represented as $I'(\kappa.\mathtt{m}) = \hat{out}$ for every $\kappa.\mathtt{m}$. Formula $\phi_{\mathtt{Subs.foreign}}$ is in Figure 2. Then $I''(\mathtt{Subs.foreign}) = ((\check{this} \to \hat{this}) \wedge \hat{out} \wedge \hat{temp}) \circ (\phi_1 \circ (\phi_2 \vee \phi_3[I \mapsto I'])) \circ (\check{out} \to \hat{out})$. To compute $\phi_3[I \mapsto I']$, plug $\hat{out}$ instead of each occurrence of I. We have $\phi_3[I \mapsto I'] = (((\check{temp} \to \hat{this}) \circ \hat{out} \circ (\check{out} \to \hat{res})) \wedge (\check{out} \to \hat{out})) \circ ((\check{res} \to \hat{out}) \wedge (\check{this} \to \hat{this}) \wedge (\check{temp} \to \hat{temp}))$. We show an example of (associative) composition: $\hat{out} \circ (\check{out} \to \hat{res}) = \exists_{out}\,(out' \wedge (out' \to \hat{res})) = (out' \wedge (out' \to \hat{res}))[out' \mapsto true] \vee (out' \wedge (out' \to \hat{res}))[out' \mapsto false] = \hat{res}$. Continuing the calculation we have $\phi_3[I \mapsto I'] = (\hat{res} \wedge (\check{out} \to \hat{out})) \circ ((\check{res} \to \hat{out}) \wedge (\check{this} \to \hat{this}) \wedge (\check{temp} \to \hat{temp})) = \hat{out}$. It can also be checked that $\phi_1 \circ (\phi_2 \vee \phi_3[I \mapsto I']) = \hat{out}$. Hence $I''(\mathtt{Subs.foreign}) = ((\check{this} \to \hat{this}) \wedge \hat{temp} \wedge \hat{out}) \circ \hat{out} \circ (\check{out} \to \hat{out}) = \hat{out}$. The same can be computed for `ForeignSubs.foreign`. In conclusion $I' = I''$ is the least fixpoint of the transformer of Definition 15 (compare with Example 11).

8 Conclusion

We have applied abstract compilation into Boolean formulas to define a static analysis which detects non-cyclicity of program variables. This leads to an elegant and relational formulation of the analysis, and in prospective to an implementation through *binary decision diagrams* [4]. This is considered very important for building efficient implementations of static analyses [2].

References

1. K. R. Apt and D. Pedreschi. Reasoning about Termination of Pure Prolog Programs. *Information and Computation*, 106(1):109–157, September 1993.
2. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
3. A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *Journal of Logic Programming*, 19/20:149–197, 1994.

4. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
5. P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992.
6. M. Hermenegildo, W. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(2 & 3):349–366, 1992.
7. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions (extended version with proofs). Available at `http://www.sci.univr.it/∼spoto/papers.html`, 2005.
8. S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, 2005.
9. Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. In *Principles of Programming Languages (POPL)*, pages 32–41, St. Petersburg Beach, Florida, USA, January 1996.
10. R. Wilhelm, T. W. Reps, and S. Sagiv. Shape Analysis and Applications. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook*, pages 175–218. CRC Press, 2002.
11. G. Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

Efficient Strongly Relational Polyhedral Analysis*

Sriram Sankaranarayanan[1,3], Michael A. Colón[2],
Henny Sipma[3], and Zohar Manna[3]

[1] NEC Laboratories America, Princeton, NJ
sriramsQnec-labs.com
[2] Center for High Assurance Computer Systems, Naval Research Laboratory
colon@itd.nrl.navy.mil
[3] Computer Science Department, Stanford University, Stanford, CA 94305-9045
(sipma, zm)@theory.stanford.edu

Abstract. Polyhedral analysis infers invariant linear equalities and inequalities of imperative programs. However, the exponential complexity of polyhedral operations such as image computation and convex hull limits the applicability of polyhedral analysis. Weakly relational domains such as intervals and octagons address the scalability issue by considering polyhedra whose constraints are drawn from a restricted, user-specified class. On the other hand, these domains rely solely on candidate expressions provided by the user. Therefore, they often fail to produce strong invariants.

We propose a polynomial time approach to strongly relational analysis. We provide efficient implementations of join and post condition operations, achieving a trade off between performance and accuracy. We have implemented a strongly relational polyhedral analyzer for a subset of the C language. Initial experimental results on benchmark examples are encouraging.

1 Introduction

Polyhedral analysis seeks to discover invariant linear equality and inequality relationships among the variables of an imperative program. The computed invariants are used to establish safety properties such as freedom from buffer overflows. The standard approach to polyhedral analysis is through a fixed point iteration in the domain of convex polyhedra [9]. Complexity considerations, however, restrict its application to small systems. Libraries such as NewPolka [13] and PPL [2] have made strides towards addressing some of these tractability issues, but still the approach remains impractical for large systems.

At the heart of this intractability lies the need to repeatedly convert between constraint and generator representations of polyhedra. Efficient analysis techniques work on restricted forms of polyhedra wherein such a conversion can

* This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363 and CCF-0430102, by ARO grant DAAD 19-01-1-0723 and by NAVY/ONR contract N00014-03-1-0939 and the Office of Naval Research.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 111–125, 2006.
© Springer-Verlag Berlin Heidelberg 2006

be avoided. *Weakly relational domains* such as octagons [17], intervals [7], octahedra [6] and the TCM domain [18], avoid these conversions by considering polyhedra whose constraints are fixed a priori. The abstract domain of Simon et al. [19] considers polyhedra with at most two variables per constraint. Using these syntactic restrictions, the analysis can be carried out efficiently. However, the main drawback of such syntactic restrictions is the inability of the analysis to infer invariants that require expressions of an arbitrary form. Thus, in many cases, such domains may fail to prove the property of interest.

In this paper, we provide an efficient strongly relational polyhedral domain by drawing on ideas from both weak and strong relational analysis. We present alternatives to the join and post condition operations. In particular, we provide a new join algorithm, called *inversion join*, that works in polynomial time in the size of the input polyhedra, as opposed to the exponential space polyhedral join. We make use of linear programming to implement an efficient join and post condition operators, along with efficient inclusion checks and widening operators.

On the other hand, our domain operations are weaker than the conventional polyhedral domain operations, potentially yielding weaker invariants. Using a prototype implementation of our techniques, we have analyzed several sorting and string handling routines for buffer overflows. Our initial results are promising; our analysis performs better than the standard approaches while computing invariants that are sufficiently strong in practice.

Outline. Section 2 discusses the preliminary notions of polyhedra, transition systems and invariants. Section 3 discusses algorithms for domain operations needed for polyhedral analysis. Section 4 discusses the implementation and the results obtained on benchmark examples.

2 Preliminaries

We recall some standard results on polyhedra, followed by a brief description of system models and abstract interpretation. Throughout the paper, let $\mathcal{R}$ represent the set of reals and $\mathcal{R}^+ = \mathcal{R} \cup \{\pm\infty\}$ represent the extended real numbers.

Definition 1 (Linear Assertions). *A* linear expression e *is of the form* $a_1x_1 + \cdots + a_nx_n + b$*, wherein each* $a_i \in \mathcal{R}$ *and* $b \in \mathcal{R}^+$*. The expression is said to be* homogeneous *if* $b = 0$*. A* linear constraint *is of the form* $a_1x_1 + \cdots + a_nx_n + b \bowtie 0$*, with* $\bowtie \in \{\geq, =\}$*. A* linear assertion *is a finite conjunction of linear inequalities.*

Note that the linear inequality $e + \infty \geq 0$ represents the assertion *true*, whereas the inequality $e - \infty \geq 0$ represents *false*. Since each equality $e = 0$ can be represented as a conjunction of two inequalities, an assertion can be written in matrix form as $A\vec{x} + \vec{b} \geq \vec{0}$, where A is a $m \times n$ matrix, while $\vec{x}$ and $\vec{b}$ are n and m-dimensional vectors, respectively. The set of points in $\mathcal{R}^n$ satisfying a linear assertion is called a *polyhedron*.

The representation of a polyhedron by a linear assertion is known as its *constraint representation*. Alternatively, a polyhedron may be represented explicitly

by a finite set of vertices and rays, known as the *generator representation*. Each representation may be exponentially larger than the other. For instance, the n dimensional hypercube is represented by $2n$ constraints and 2^n generators. Efficient libraries of conversion algorithms such as the new PolKa [12] and the Parma Polyhedral Library (PPL) [2] have made significant improvements to the size of the polyhedra for which the conversion is possible. Nevertheless, this conversion still remains intractable for large polyhedra involving 100s of variables and constraints.

A *Template Constraint Matrix* (TCM) T is a finite set of homogeneous linear expressions over $\vec{x}$. Given an assertion φ, its expressions induce a TCM T which we shall denote as $\mathsf{Ineqs}(\varphi)$. If φ is represented as $A\vec{x}+\vec{b} \geq 0$ then $\mathsf{Ineqs}(\varphi) : A\vec{x}$.

Linear Programming. We briefly describe the theory of linear programming. Details may be found in standard textbooks [5].

Definition 2 (Linear Programming). *A canonical instance of the* linear programming (LP) problem *is of the form*

$$\mathsf{minimize}\ e\ \mathsf{subject\ to}\ \varphi\,,$$

for assertion φ and a linear expression e, called the objective function.

The goal is to determine the solution of φ for which e is minimal. A LP problem can have one of three results: (1) an optimal solution; (2) $-\infty$, i.e, e is unbounded from below in φ; (3) $+\infty$, i.e, φ has no solutions.

It is well-known that an optimal solution, if it exists, is realized at a vertex of the polyhedron. Therefore, the optimal solution can be found by evaluating e at each of the vertices. Enumerating all the vertices is very inefficient because the number of generators is worst-case exponential in the number of constraints. The popular SIMPLEX algorithm (due to Danzig [10]) employs a sophisticated hill climbing strategy that converges on an optimal vertex without necessarily enumerating all vertices. In theory, the technique is worst-case exponential. The SIMPLEX method is efficient over most problems. Interior point methods such as *Karmarkar's* algorithm and other techniques based on ellipsoidal approximations are guaranteed to solve linear programs in polynomial time. Using an open-source implementation of SIMPLEX such as GLPK [15], massive LP instances involving tens of thousands (10^4 and beyond) of variables and constraints can be solved efficiently.

Programs and Invariants

We assume programs over real valued variables without any function calls. The program is represented by a linear transition system also known as a *control flow graph.*

Definition 3 (Linear Transition Systems). *A* linear transition system *(LTS) $\Pi : \langle L, \mathcal{T}, \ell_0, \Theta\rangle$ over a set of variables V consists of*

– *L: a set of locations (cutpoints);*
– *$\mathcal{T}$: a set of transitions (edges), where each transition $\tau : \langle \ell_i, \ell_j, \rho_\tau \rangle$ consists of a pre-location ℓ_i, a post-location ℓ_j, and a transition relation ρ_τ, represented as a linear assertion over $V \cup V'$, where V denotes the values of the variables in the current state, and V' their values in the next state;*
– *$\ell_0 \in L$: the initial location;*
– *Θ: a linear assertion over V specifying the initial condition.*

A *run* of a LTS is a sequence $\langle m_0, s_0 \rangle, \langle m_1, s_1 \rangle, \ldots$, with $m_i \in L$ and s_i a valuation of V, also called a *state*, such that

– Initiation: $m_0 = \ell_0$, and $s_0 \models \Theta$
– Consecution: for all $i \geq 0$ there exists a transition $\tau : \langle \ell_j, \ell_k, \rho_\tau \rangle$ such that $m_i = \ell_j$, $m_{i+1} = \ell_k$, and $\langle s_i, s_{i+1} \rangle \models \rho_\tau$.

A state s is *reachable* at location ℓ if $\langle \ell, s \rangle$ appears in some run.

A given linear assertion ψ is a *linear invariant* of a linear transition system (LTS) at a location ℓ iff it is satisfied by every state reachable at ℓ. An *assertion map* associates each location of a LTS to a linear assertion. An assertion map η is *invariant* if $\eta(\ell)$ is an invariant, for each $\ell \in L$. In order to prove a given assertion map invariant, we use the inductive assertions method due to Floyd (see [16]).

Definition 4 (Inductive Assertion Maps). *An assertion map η is inductive iff it satisfies the following conditions:*

Initiation: $\Theta \models \eta(\ell_0)$,
Consecution: *For each transition $\tau : \langle \ell_i, \ell_j, \rho_\tau \rangle$, $(\eta(\ell_i) \ \wedge \ \rho_\tau) \models \eta(\ell_j)'$. Note that $\eta(\ell_j)'$ refers to $\eta(\ell_j)[V|V']$ with variables in V substituted by their corresponding primed variables in V'.*

It is well known that any inductive assertion map is invariant. However, the converse need not be true. The standard technique for proving an assertion invariant is to find an inductive assertion that strengthens it.

Linear Relations Analysis

Linear relation analysis seeks an inductive assertion map for the input program, labeling each location with a linear assertion. Analysis techniques are based on the theory of *Abstract Interpretation* [8] and specialized for linear relations by Cousot and Halbwachs [9]. The technique starts with an initial assertion map, and weakens it iteratively using the *post*, *join* and the *widening* operators. When the iteration converges, the resulting map is guaranteed to be inductive, and hence invariant. Termination is guaranteed by the design of the widening operator.

The *post condition* operator takes an assertion φ and a transition τ, and computes the set of states reachable by τ from a state satisfying φ. It can be expressed as

$$post(\varphi, \tau) : \ (\exists V_0)(\varphi(V_0) \ \wedge \ \rho_\tau(V_0, V))$$

Standard polyhedral operations can be used to compute post. However, more efficient strategies for computing post exist when ρ_τ has a special structure. Given assertions $\varphi_{\{1,2\}}$ such that $\varphi_1 \models \varphi_2$, the *standard widening* $\varphi_1 \nabla \varphi_2$ is an assertion φ that contains all the inequalities in φ_1 that are satisfied by φ_2. The details along with key mathematical properties of widening are described in [9, 8], and enhanced versions appear in [12, 4, 1]. As mentioned earlier, the analysis begins with an initial assertion map defined by $\eta_0(\ell_0) = \Theta$, and $\eta_0(\ell) = \mathit{false}$ for $\ell \neq \ell_0$. At each *step*, the map η_i is updated to map η_{i+1} as follows:

$$\eta_{i+1}(\ell) = \eta_i(\ell) \;\langle \text{OP} \rangle \left[\eta_i(\ell) \bigsqcup_{\tau_j \equiv \langle \ell_j, \ell, \rho \rangle} (post(\eta_i(\ell_j), \tau_j)) \right] ,$$

where OP is the join ($\sqcup$) operator for a propagation step, and the widening (∇) operator for a widening step. The overall algorithm requires a predefined *iteration strategy*. A typical strategy carries out a fixed number of initial propagation steps, followed by widening steps until termination.

Linear Assertion Domains

Linear relation analysis is performed using a forward propagation wherein polyhedra are used to represent sets of states. Depending on the family of polyhedra considered, such domains are classified as *weakly relational* or *strongly relational*.

Let $T = \{e_1, \ldots, e_m\}$ be a TCM. The weakly relational domain induced by T consists of assertions $\bigwedge_{e_i \in T} e_i + b_i \geq 0$ for $b_i \in \mathcal{R}^+$. TCMs and their induced weakly relational domain are formalized in our earlier work [18]. Given a weakly relational domain defined by a TCM T and a linear transition system Π, we seek an inductive assertion map η such that $\eta(\ell)$ belongs to the domain of T for each location ℓ. Many weakly relational domains have been studied: Intervals, octagons and octahedra are classical examples.

Example 1 (Weakly Relational Analysis). Let X be the set of system variables. The interval domain is defined by the TCM consisting of expressions $T_X = \{\pm x_i \mid x_i \in X\}$. Thus, any polyhedron belonging to the domain is an interval expression of the form $\bigwedge (x_i + a_i \geq 0 \;\wedge\; -x_i + b_i \geq 0)$. The goal of interval analysis is to discover the coefficients $a_i, b_i \in \mathcal{R}^+$ representing the bounds for each variable x_i at each location of the program [7].

The octagon domain of Miné subsumes the interval domain by considering additional expressions of the form $\pm x_i \pm x_j$ such that $x_i, x_j \in X$ [17]. The octahedron domain due to Clarisó and Cortadella considers expressions of the form $\sum_i a_i x_i$ such that $a_i \in \{-1, 0, 1\}$ [6].

It is possible to carry out the analysis in any weakly relational domain efficiently [18].

Theorem 1. *Given a TCM T and a linear system Π, all the domain operations for the weakly relational analysis of Π in the domain induced by T can be performed in time polynomial in $|T|$ and $|\Pi|$.*

```
integer x,y where (x = 1 ∧ x ≥ y)
ℓ0 : while true do
    if (x ≥ y) then
      (x, y) := (x + 2, y + 1)
    else
      (x, y) := (x + 2, y + 3)
    end if
  end while
```

Fig. 1. An example program

Weakly relational domains are appealing since the analysis in these domains is scalable to large systems. On the other hand, the invariants they produce are often imprecise. For instance, even if $e_i + a_i \geq 0$ is invariant for some expression e_i in the TCM, its proof may require an inductive strengthening $e_j + a_j \geq 0$, where e_j is not in the TCM.

A *strongly relational* analysis does not syntactically restrict the polyhedra considered. The polyhedral domain is not restricted in its choice of invariant expressions, and is potentially more precise than a weakly relational domain. The main drawback, however, is the high complexity of the domain operations. Each domain operation requires conversions from the constraint to the generator representation and back. Popular implementations of strongly relational analysis require worst case exponential space due to repeated representation conversions.

Example 2. Consider the system in Figure 1. Interval and octagon domains both discover the invariant $\infty \geq x \geq 1$ at location ℓ_0. A strongly relational analysis such as polyhedral analysis discovers the invariant $x \geq 1 \ \wedge \ 3x - 2y \geq 1$, as does the technique that we present.

3 Domain Operations

The theory of Abstract Interpretation provides a framework for the design of program analyses. A sound program analysis can be designed by constructing an abstract domain with the following domain operations:

Join (union). Given two assertions φ_1, φ_2 in the domain, we seek an assertion φ such that $\varphi_1 \models \varphi$ and $\varphi_2 \models \varphi$. In many domains, it is possible to find the strongest possible φ satisfying this condition. The operation of computing such an assertion is called the *strong join*.

Post Condition. Given an assertion φ, and a transition relation ρ_τ, we seek an assertion ψ such that $\varphi[V] \ \wedge \ \rho_\tau[V, V'] \models \psi[V']$. A *strong post condition* operator computes the strongest possible assertion ψ satisfying this condition.

Widening. Widening ensures the termination of the fixed point iteration.

Additionally, inclusion tests between assertions are important for detecting the termination of an iteration. Feasibility tests and redundancy elimination are also

frequently used to speed up the analysis. We present several different join and post condition operations, each achieving a different trade off between efficiency and precision.

Join

Given two linear assertions φ_1 and φ_2 over a vector $\vec{x}$ of system variables, we seek a linear assertion φ, such that both $\varphi_1 \models \varphi$ and $\varphi_2 \models \varphi$.

Strong Join. The strong join seeks the strongest assertion φ (denoted $\varphi_1 \sqcup_s \varphi_2$) subsuming both φ_1 and φ_2. In the domain of convex polyhedra, this is known as the *polyhedral convex hull* and is obtained by computing the generator representations of φ_1 and φ_2. The set of generators of φ is the union of those of φ_1 and φ_2. This representation is then converted back into the constraint representation. Due to the repeated representation conversions, the strong join is worst-case exponential space in the size of the input assertions.

Example 3. Consider the assertions

$$\begin{aligned} \varphi_1 &: x - y \leq 5 \wedge y + x \leq 10 \wedge -10 \leq\ x \leq 5 \\ \varphi_2 &: x - y \leq 9 \wedge y + x \leq 5 \ \ \wedge -9 \leq\ x \leq 6 \end{aligned}$$

Their strong join $\varphi_1 \sqcup_s \varphi_2$, generated by the union of their vertices, is

$$\varphi:\ 6x + y \leq 35\ \wedge\ y + 3x + 45 \geq 0\ \wedge\ x - y \leq 9\ \wedge\ x + y \leq 10\ \wedge\ -10 \leq x \leq 6\,.$$

Weak Join. The weak join operation is inspired by the join used in weakly relational domains.

Definition 5 (Weak Join). *The weak join of two polyhedra φ_1, φ_2 is computed as follows:*

1. Let TCM $T = \mathsf{Ineqs}(\varphi_1) \cup \mathsf{Ineqs}(\varphi_2)$ be the set of inequality expressions that occur in either of $\varphi_{\{1,2\}}$. Recall that each equality in φ_1 or φ_2 is represented by two inequalities in T.
2. For each expression e_i in T, we compute the values a_i and b_i using linear programming, as follows:

$$\begin{aligned} a_i &= \mathsf{minimize}\ e_i\ \mathsf{subject\ to}\ \varphi_1 \\ b_i &= \mathsf{minimize}\ e_i\ \mathsf{subject\ to}\ \varphi_2 \end{aligned}$$

It follows that $\varphi_1 \models (e_i \geq a_i)$ and $\varphi_2 \models (e_i \geq b_i)$.
3. Let $c_i = \min(a_i, b_i)$. Therefore, both $\varphi_1, \varphi_2 \models (e_i \geq \min(a_i, b_i) \equiv c_i)$.
The weak join $\varphi_1 \sqcup_w \varphi_2$ is given by the assertion $\bigwedge_{e_i \in T}\ e_i \geq c_i$.

The advantage of the weak join is its efficiency: it can be computed using LP queries, where both the number of such queries and the size of each individual query is polynomial in the input size. On the other hand, the weak join does not discover any new relations. It is weaker than the strong join, as shown by the argument above (and strictly so, as shown by the following Example).

Example 4. Consider the assertions φ_1, φ_2 from Example 3 above. The TCM T and the a_i, b_i values are shown in the table below:

T :

#	*Relation*		$a_i(\varphi_1)$	$b_i(\varphi_2)$
1	$y - x$	$\geq$	-5	-9
2	$-y - x$	$\geq$	-10	-5
3	x	$\geq$	-10	-9
4	$-x$	$\geq$	-5	-6

The weak join is given by

$$\varphi_w : \; (y - x \geq -9 \;\wedge\; -y - x \geq -10 \;\wedge\; x \geq -10 \;\wedge\; -x \geq -6)\,.$$

This result is strictly weaker than the strong join computed in Example 3.

Restricted Joins. The weak join shown above is more efficient than the strong join. However, this efficiency comes at the cost of precision. We therefore seek efficient alternatives to strong and weak join. The k-restricted join (denoted $\sqcup_k$) improves upon the weak join as follows:

1. Choose a subset of inequalities from φ_1, φ_2, each of cardinality at most k. Let ψ_1 and ψ_2 be the assertions formed by the chosen inequalities. In general, ψ_1, ψ_2 may contain different sets of inequalities, even different cardinalities. Note that $\varphi_i \models \psi_i$ for $i = 1, 2$.
2. Compute the strong join $\psi_1 \sqcup_s \psi_2$ in *isolation*. Conjoin the results with the weak join $\varphi_1 \sqcup_w \varphi_2$.
3. Repeat step 1 for a different set of choices of $\psi_{\{1,2\}}$, while conjoining each such join to the weak join.

Since $\varphi_i \models \psi_i$, for $i = 1, 2$, it follows by the monotonicity of the strong join operation that $\varphi_1 \sqcup_s \varphi_2 \models \psi_1 \sqcup_s \psi_2$. Thus $\varphi_1 \sqcup_s \varphi_2 \models \varphi_1 \sqcup_k \varphi_2$ for each $k \geq 0$.

Let φ_1, φ_2 have at most m constraints. The k-restricted join requires vertex enumeration for $O(\binom{m}{k}^2)$ polyhedra with at most k constraints. As such, this join is efficient only if k is a small constant. We shall now provide an efficient $O(m^2)$ algorithm based on $\sqcup_2$, to improve the weak join.

Inversion Join. The inversion join is based on the 2-restricted join. Let T be the TCM and a_i, b_i be the values computed for the weak join as in Definition 5. Consider pairs of expressions $e_i, e_j \in T$ yielding the assertions

$$\begin{array}{l} \psi_1 : e_i \geq a_i \wedge e_j \geq a_j \\ \psi_2 : e_i \geq b_i \wedge e_j \geq b_j \end{array}$$

We use the structure of the assertions ψ_1, ψ_2 to perform their strong join analytically. The key notion is that of an *inversion*.

Definition 6 (Inversion). *Expressions $e_i, e_j \in T$ and corresponding coefficients a_i, a_j, b_i, b_j form an* inversion *iff the following conditions hold:*

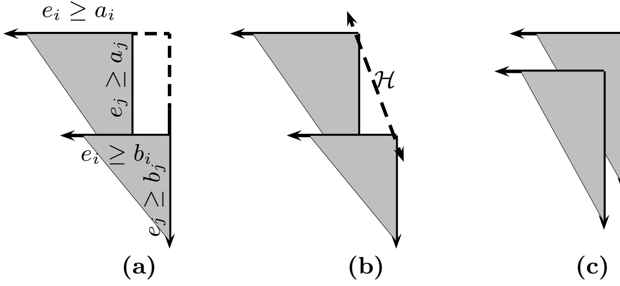


Fig. 2. (a) $a_i > b_i,\ a_j < b_j$, (b) Weak join is strictly weaker than strong join, (c) $a_i > b_i,\ a_j > b_j$: Weak join is the same as strong join

1. *$a_i, a_j, b_i, b_j \in \mathcal{R}$, i.e, none of them is $\pm\infty$.*
2. *$e_i \neq \lambda e_j$ for $\lambda \in \mathcal{R}$, i.e, e_i, e_j are linearly independent.*
3. *$a_i < b_i$ and $b_j < a_j$ (or vice-versa).*

Example 5. Consider two "wedges" $\psi_1 :\ e_i \geq a_i \ \wedge\ e_j \geq a_j$ and $\psi_2 :\ e_i \geq b_i \ \wedge\ e_j \geq b_j$. Depending on the values of a_i, a_j, b_i, b_j, two cases arise as depicted in Figures 2(a,b,c). Figures 2(a,b) form an inversion. When this happens, the weak join (a) is strictly weaker than the strong join (b). Figure 2(c) does not form an inversion. The weak and strong joins coincide in this case.

Therefore, a strong join of polyhedra that form an inversion gives rise to a half space that is not discovered by the weak join. We now derive this "missing half-space" $\mathcal{H}$ analytically.

The half space subsumes both ψ_1 and ψ_2. A half-space that is a consequence of $\psi_1 :\ e_i \geq a_i \ \wedge\ e_j \geq a_j$ is of the form $\mathcal{H} :\ e_i + \lambda_{ij} e_j \geq a_i + \lambda_{ij} a_j$, for some $\lambda_{ij} \geq 0$. Similarly for ψ_2, we obtain $\mathcal{H} :\ e_i + \lambda_{ij} e_j \geq b_i + \lambda_{ij} b_j$. Equating coefficients, yields the equation $a_i + \lambda_{ij} a_j = b_i + \lambda_{ij} b_j$. The required value of λ_{ij} is

$$\lambda_{ij} = \frac{a_i - b_i}{b_j - a_j}\,.$$

Note that requiring $\lambda_{ij} > 0$ yields $a_i < b_i$ and $b_j < a_j$. Therefore, ψ_1, ψ_2 contain a non trivial common half-space iff they form an inversion.

Definition 7 (Inversion Join). *Given φ_1, φ_2 the inversion join $\varphi_1 \sqcup_{inv} \varphi_2$ is computed as follows:*

1. *Compute the TCM $T = \mathsf{Ineqs}\,(\varphi_1) \cup \mathsf{Ineqs}\,(\varphi_2)$.*
2. *For each $e_i \in T$ compute a_i, b_i as defined in Definition 5 using linear programming. At this point $\varphi_1 \models e_i \geq a_i$ and $\varphi_2 \models e_i \geq b_i$. Let $\varphi_w = \varphi_1 \sqcup_w \varphi_2$ be the weak join.*
3. *For each pair e_i, e_j, consider the expression $e_i + \lambda_{ij} e_j \geq a_i + \lambda_{ij} a_j$, with λ_{ij} as defined above.*

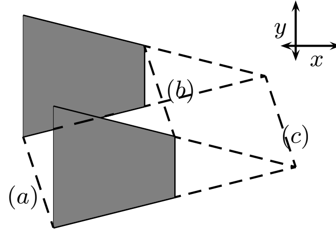


Fig. 3. Inversion join over two polyhedra (a), (b) and (c) are the newly discovered relations

4. *The inversion join is the conjunction of φ_w and all the inversion expressions generated in Step 3. Optionally, simplify the result by removing redundant inequalities.*

Example 6. Figure 3 shows the result of an inversion join over two input polyhedra φ_1, φ_2 used in Example 3. Example 4 shows the TCM T and the a_i, b_i values. There are three inversions

#	Expressions	Subsuming Half-Space
(a)	$\langle 1, 3 \rangle$	$y + 3x + 45 \geq 0$
(b)	$\langle 2, 4 \rangle$	$-y - 6x + 35 \geq 0$
(c)	$\langle 1, 2 \rangle$	$y - 9x + 65 \geq 0$

The "expressions" column in the table above refers to expressions by their row numbers in the table of Example 4. From Figure 3, note that (c) is redundant. Therefore the result of the join may require redundancy elimination (algorithm provided later in this section). This result is equivalent to the result of the strong join in Example 3.

Theorem 2. *Let φ_1, φ_2 be two polyhedra. It follows that*

$$\varphi_1 \sqcup_s \varphi_2 \models \varphi_1 \sqcup_{inv} \varphi_2 \models \varphi_1 \sqcup_w \varphi_2 \,.$$

The inversion join requires as many LP queries as the weak join and additional $O(m^2 n)$ arithmetic operations to compute inversions, where m is the number of inequalities in T and n, the dimensionality.

Note. The descriptions of the weak and inversion join treat each equality as two inequalities. The resulting join could be made more precise if additionally, the equality join defined by Karr's analysis [14] is computed and conjoined to the result. This can be achieved in time that is polynomial in the number of equalities.

Post Condition

The post condition computes the image of an assertion φ under a transition relation of the form $\xi \ \wedge \ \vec{x}' = A\vec{x} + \vec{b}$. This is equivalent to the image of

$\varphi \wedge \xi$ under the affine transformation $\vec{x}' = A\vec{x} + \vec{b}$. If the matrix A is invertible, then this image is easily computed by substituting $\vec{x} = A^{-1}(\vec{x}' - \vec{b})$ [9]. On the other hand, it is frequently the case that A is not invertible. We present three alternatives, the strong, weak and restricted post conditions.

Strong Post. The strong post is computed by first enumerating the generators of $\varphi \wedge \xi$. Each generator is transformed under the operation $A\vec{x} + \vec{b}$. The resulting polyhedron is generated by these images. Conversion back to the constraint representation completes the computation.

Weak Post. Weak post requires a TCM T' labeling the post location of the transition. Alternatively, this TCM may be derived from $\mathsf{Ineqs}\,(\eta(\ell'))$ where $\eta(\ell')$ labels the post-location. Given the existence of such a TCM, we may use the post operation defined for TCMs [18] to compute the weak post.

Note. The post condition computation for equalities can be performed separately using the image operation defined for Karr's analysis. This can be added to the result, thus strengthening the weak post.

k-Restricted Post The k-restricted post condition improves upon the weak post by using the monotonicity of the strong post operation (see [8]) similar to the k-restricted join algorithm. Therefore, considering a subset of up to k inequalities ψ, we may compute the strong post of ψ and add the result conjunctively to the weak post. The results improve upon the precision of weak post. As is the case for join, it is possible to treat the cases for $k = 1, 2$ efficiently.

Example 7. Consider the polyhedron $\varphi :\ x - y \geq 0\ \wedge\ x \leq 0\ \wedge\ x + y + 3 \leq 0$ and the transformation $x := x + 3, y := 0$. Consider the TCM $\mathrm{T} = \{x - y, x + y, y - x, -x - y\}$. The weak post of φ w.r.t T is computed by finding bounds for each expression. For instance the bound for $x - y$ is discovered by solving:

$$\mathsf{minimize}\ x' - y'\ \mathsf{s.t.}\ \varphi\ \wedge\ x' = x + 3\ \wedge\ y' = 0$$

The overall weak post is obtained by solving 4 LPs, one for each element of T,

$$\varphi_w :\ 3 \geq x - y \geq 1.5\ \wedge\ 3 \geq x + y \geq 1.5\,.$$

This is strictly weaker than the strong post $\varphi_s :\ 3 \geq x \geq 1.5\ \wedge\ y = 0$. The 1-restricted post computes the post condition of each half-space in φ separately. This yields the result $y = 0$ for all the three half-spaces. Conjoining the 1-restricted post with the weak post yields the same result as the strong post in this example.

Note. The projection operation, an important primitive for interprocedural analysis, can be implemented along the same lines as the post condition operation, yielding the strong, weak and restricted projection operations.

Feasibility, Inclusion Check and Redundancy Elimination

There exist polynomial time algorithms that are efficient in practice for checking feasibility of a polyhedron and inclusion between two polyhedra.

Feasibility. The SIMPLEX method can be used to check feasibility of a given linear inequality assertion φ. In practice, we solve the optimization problem minimize 0 subject to φ. An answer of $+\infty$ indicates the infeasibility of φ.

Inclusion Check. As a primitive, consider the problem of checking whether a given inequality $e \geq 0$ is entailed by φ, posing the LP: minimize e subject to φ. If the optimal solution is a, it follows from the definition of a LP problem that $\varphi \models e \geq a$. Thus subsumption holds iff $a \geq 0$. In order to decide if $\varphi \models A\vec{x} + \vec{b} \geq 0$, we decide if the entailment holds for each half-space $A_i\vec{x} + b_i \geq 0$.

Redundancy Elimination (Simplification). Each inequality is checked for subsumption by the remaining inequalities using the inclusion check primitive.

Widening

The standard widening of Cousot and Halbwachs may be implemented efficiently using linear programming. Let φ_1, φ_2 be two polyhedra such that $\varphi_1 \models \varphi_2$.

Let us assume that φ_1, φ_2 are both satisfiable. We seek to drop any constraint $e_i \geq 0$ in φ_1 that is not a consequence of φ_2. This can be achieved by the inclusion test primitive described above.

Definition 8 (Standard Widening). *The standard widening of two polyhedra $\varphi_1 \models \varphi_2$, denoted $\varphi = \varphi_1 \nabla \varphi_2$ is computed as follows,*

1. *Check satisfiability of φ_1, φ_2. If either one is unsatisfiable, widening reduces to their join.*
2. *Otherwise, for each $e_i \in$ Ineqs(φ_1), compute $b_i =$ minimize e_i subject to φ_2. If $b_i < 0$ then drop the constraint $e_i \geq 0$ from φ_1.*

This operator is identical to the widening defined by Cousot and Halbwachs [9]. The operator may be improved by additionally computing the join of the equalities in both polyhedra. The work of Bagnara et al. [1] presents several approaches to improving the precision of widening operators.

4 Performance

We have implemented many of the ideas in this paper in the form of an abstract domain library written in OCAML. Our library uses GLPK [15] to solve LP queries, and PPL [2] to convert between the constraint and generator representations of polyhedra. Such conversions are used to implement the strong join and post condition. Communication between the different libraries is implemented using Unix pipes. As a result, the communication overhead is significant for small examples.

Choosing Domain Operations. We have provided several options for the join and the post condition operations. In practice, one can envision many strategies for choosing among these operations. Our implementation chooses between the strong and the weak versions based on the sizes of the input polyhedra. Strong post condition and joins are used for smaller polyhedra (40 variables+constraints). On the

other hand, the inversion join is used for polyhedra with roughly 100s of variables+constraints, while the weak versions are used for larger polyhedra. We observe empirically that the use of strong operations does not improve the result once the widening phase is started. Therefore, we resort to weak join and post condition for the widening phase of the analysis.

4.1 Benchmark Examples

We supplied our library to generate invariants for a few benchmark system models drawn from related projects such as FAST [3] and our previous work [18]. Table 1 shows the complexity of each system in terms of number of variables (#vars) along with the performance of our technique of mixed strong, weak and inversion domain operations as compared with the purely strong join/post operations implemented directly in C++ using the PPL library. We compare the running time and memory utilization of both implementations. Results were measured on an Intel Xeon II processor with 1GB RAM. The last column compares the invariants generated. A "+" indicates that our technique discovers strictly stronger invariants whereas a "≠" denotes that the invariants are incomparable.

Also, for small polyhedra, strong operations frequently outperform weak domain operations in terms of time. However, their memory consumption seems asymptotically exponential. Therefore, weak domain operations yield a drastic performance improvement when the size of the benchmark examples increases beyond the physical memory capacity of the system. Comparing the invariants generated, it is interesting to note that the invariants produced by both techniques are, for the most part, incomparable. While inversion join is weaker than strong join, the non-monotonicity of the widening operation and its dependence on the syntactic representation of the polyhedra cause the two versions to compute different invariants.

Analysis of πVC Programs. We applied our abstract domain library to analyze a subset of the *C* language called *πVC*, consisting of imperative programs over

Table 1. Performance on Benchmark Examples. All times are in seconds and memory utilization in Mbs.

Name (#vars)	#trans	Strong+Weak		Purely Strong		±
		time	mem	time	mem	
REQ-GRANT(11)	8	3.14	5.7	0.1	4.1	+
CSM(13)	8	6.21	5.9	0.1	4.2	≠
C-PJAVA(18)	14	11.2	6.0	0.1	4.1	≠
MULTIPOOL(18)	21	10.0	6.0	2.1	9.2	+
INCDEC(32)	28	39.12	6.8	8.7	10.4	≠
MESH2X2(32)	32	33.8	6.4	18.53	66.2	≠
BIGJAVA(44)	37	46.9	7.2	256.2	55.3	≠
MESH3X2(52)	54	122	8.1	$> 1h+$	$> 800+$	+

integers with function calls. The language features dynamically allocated arrays, records and recursive function calls while excluding pointers. Parameters are passed by value and global variables are disallowed. The language incorporates invariant annotations by the user that are verified by the compiler using a background decision procedure. Our analysis results in sound annotations that aid the verifying compiler in checks for runtime safety such as freedom from overflows, and with optional user supplied assertions, help prove functional correctness.

Our analyzer is inter-procedural, using summaries to handle function calls in a context sensitive manner. Our abstraction process models arrays in terms of their allocated sizes while treating their contents as unknowns. Integer operations such as multiplication, division and modulo are modeled conservatively so that soundness is maintained. The presence of recursive function calls requires that termination be ensured by limiting the number of summary instances per function and by widening on the summary preconditions.

Table 2 shows the performance on implementations of standard sorting algorithms, string search algorithms and a part of the `web2C` code for converting Pascal-style writes into C-style printf functions, originally verified by Dor et al. [11]. The columns in Table 2 show the size of each program in lines of code and number of functions. An asterisk (*) identifies programs containing recursive functions. We place a check mark ($\surd$) in the "proves property" column if the resulting annotations themselves prove all array accesses and additional user provided assertions. Otherwise, the number of unproven accesses/assertions is indicated. Our analyzer proves a vast majority ($\geq 90\%$) of the assertions valid, without any user interaction. Indirect array accesses such as $a[b[i]]$ are a major reason for the false positives. We are looking into more sophisticated abstractions to handle such accesses. The invariants generated by both the versions are similar for small programs, even though weak domain operations were clearly used during the analysis. The difference in performance is clearer as the size of the program increases. Our interface to the PPL library represents coefficients using long integers. This led to an overflow error while analyzing quicksort.

In conclusion, we have designed and implemented efficient domain operations and applied our technique to verify interesting benchmark examples. We hope to extend our analyzer to handle essential features such as pointers and arrays.

Table 2. Performance of invariant generator for benchmark programs

Description	Size		(Weak+Strong)		(Strong)		Proves
	#LOC	#fns	time(sec)	mem(Mb)	time(sec)	mem(Mb)	Property
binary-search (*)	27	2	0.48	7.8	0.4	7.5	$\surd$
insertionsort	37	1	2.9	7.9	2	7.8	$\surd$
heapsort	75	5	26.2	9.8	23.0	9.6	$\surd$
quicksort (*)	106	4	2m	13.2	overflow		$\surd$
Knuth-Morris-Pratt	110	4	9.4	8.6	7.9	8.6	4
Boyer-Moore	106	3	33.7	10.4	28.8	10.8	12
fixwrites(*)	270	10	4.2m	26.5	$>$ 75m	$>$ 75M	28

Acknowledgments. Many thanks to Mr. Aaron Bradley for implementing the πVC front end, the reviewers for their incisive comments and to the developers of the PPL [2] and GLPK [15] libraries.

References

1. Bagnara, R., Hill, P. M., Ricci, E., and Zaffanella, E. Precise widening operators for convex polyhedra. In *Static Analysis Symposium* (2003), vol. 2694 of *LNCS*, Springer–Verlag, pp. 337–354.
2. Bagnara, R., Ricci, E., Zaffanella, E., and Hill, P. M. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *SAS* (2002), vol. 2477 of *LNCS*, Springer–Verlag, pp. 213–229.
3. Bardin, S., Finkel, A., Leroux, J., and Petrucci, L. FAST: Fast accelereation of symbolic transition systems. In *Computer-aided Verification* (July 2003), vol. 2725 of *LNCS*, Springer–Verlag.
4. Besson, F., Jensen, T., and Talpin, J.-P. Polyhedral analysis of synchronous languages. In *Static Analysis Symposium* (1999), vol. 1694 of *LNCS*, pp. 51–69.
5. Chvátal, V. *Linear Programming*. Freeman, 1983.
6. Clarisó, R., and Cortadella, J. The octahedron abstract domain. In *Static Analysis Symposium* (2004), vol. 3148 of *LNCS*, Springer–Verlag, pp. 312–327.
7. Cousot, P., and Cousot, R. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming* (1976), Dunod, Paris, France, pp. 106–130.
8. Cousot, P., and Cousot, R. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Principles of Programming Languages* (1977), pp. 238–252.
9. Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among the variables of a program. In *ACM POPL* (Jan. 1978), pp. 84–97.
10. Dantzig, G. B. *Programming in Linear Structures*. USAF, 1948.
11. Dor, N., Rodeh, M., and Sagiv, M. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. PLDI'03* (2003), ACM Press.
12. Halbwachs, N., Proy, Y., and Roumanoff, P. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design 11* (1997), 157–185.
13. Jeannet, B. The convex polyhedra library New Polka. Available online from `http://www.irisa.fr/prive/Bertrand.Jeannet/newpolka.html`.
14. Karr, M. Affine relationships among variables of a program. *Acta Inf. 6* (1976), 133–151.
15. Makhorin, A. The GNU Linear Programming Kit (GLPK), 2000. Available online from `http://www.gnu.org/software/glpk/glpk.html`.
16. Manna, Z. *Mathematical Theory of Computation*. McGraw-Hill, 1974.
17. Miné, A. A new numerical abstract domain based on difference-bound matrices. In *PADO II* (May 2001), vol. 2053 of *LNCS*, Springer–Verlag, pp. 155–172.
18. Sankaranarayanan, S., Sipma, H. B., and Manna, Z. Scalable analysis of linear systems using mathematical programming. In *Verification, Model-Checking and Abstract-Interpretation (VMCAI 2005)* (January 2005), vol. 3385 of *LNCS*.
19. Simon, A., King, A., and Howe, J. M. Two variables per linear inequality as an abstract domain. In *LOPSTR* (2003), vol. 2664 of *Lecture Notes in Computer Science*, Springer, pp. 71–89.

Environment Abstraction for Parameterized Verification*

Edmund Clarke[1], Muralidhar Talupur[1], and Helmut Veith[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Technische Universität München, Munich, Germany

Abstract. Many aspects of computer systems are naturally modeled as parameterized systems which renders their automatic verification difficult. In well-known examples such as cache coherence protocols and mutual exclusion protocols, the unbounded parameter is the number of concurrent processes which run the same distributed algorithm. In this paper, we introduce environment abstraction as a tool for the verification of such concurrent parameterized systems. Environment abstraction enriches predicate abstraction by ideas from counter abstraction; it enables us to reduce concurrent parameterized systems with unbounded variables to precise abstract finite state transition systems which can be verified by a finite state model checker. We demonstrate the feasibility of our approach by verifying the safety and liveness properties of Lamport's bakery algorithm and Szymanski's mutual exclusion algorithm. To the best of our knowledge, this is the first time both safety and liveness properties of the bakery algorithm have been verified at this level of automation.

1 Introduction

We propose a new method for the verification of concurrent parameterized systems which combines predicate abstraction [21] with ideas from counter abstraction [29]. In predicate abstraction, the memory state of a system is approximated by a tuple of Boolean values which indicate whether certain properties ("predicates") of the memory state hold or not. For example, instead of keeping all 64 bits for two integer variables x, y, predicate abstraction may just track the Boolean value of the predicate $x > y$.

Counter abstraction, in contrast, is specifically tailored for concurrent parameterized systems which are composed of finite state processes: for each possible state s of a single finite state process, the abstract state contains a counter C_s which denotes the number of processes currently in state s. Thus, the process identities are abstracted away in counter abstraction. It can be argued that counter abstraction constitutes a very natural abstraction mechanism for protocols. In practice, the counters in counter abstraction are themselves abstracted in that they are cut off at value 2.

Counter abstraction however has two main problems: first, it works only for finite state systems, and second, it assumes perfect symmetry, i.e., each process is identical

* This research was sponsored by the the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF. The third author was also supported by the EU GAMES Network.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 126–141, 2006.
© Springer-Verlag Berlin Heidelberg 2006

to every other process in every aspect. Well-known algorithms such as Lamport's bakery algorithm are not directly amenable to counter abstraction: the bakery algorithm has an infinite state space due to an unbounded integer variable, and also an inherent asymmetry due to its use of process *id's*.

In this paper, we will address the two disadvantages of counter abstraction by incorporating the idea of counter abstraction into a new form of predicate abstraction: *since the state space is infinite, we do not count the processes in a given state as in traditional counter abstraction, but instead we count the number of processes satisfying a given predicate.* Note that the counters which we actually use in this paper are cut off at the value 1; such degenerated counters are evidently tantamount to existential quantifiers. Counting abstraction, too, usually needs only counters in the range $[0..2]$. Since our abstraction maintains the state of one process explicitly, a range of $[0..1]$ for each counter suffices.

Our new form of abstraction is also different from common predicate abstraction frameworks: Since the number of processes in a concurrent parameterized system is *unbounded*, the system does not have a single infinite-state model, but an infinite sequence of models which increase in complexity. Moreover, since the individual processes can have local data variables with unbounded range (e.g. integers), each of these models is an infinite-state system by itself. Thus, computing the abstract transition relation is a non-trivial task. Note that the predicates need to reflect the properties of a set of concurrent processes whose cardinality we do not know at verification time. To encode the necessary information into the abstract model, we will introduce *environment predicates*.

Environment Predicates. We use an asynchronous model which crucially distinguishes between the finite control variables of a process and the unbounded data variables of a process. The control variables are used to model the finite control of the processes while the data variables can be read by other processes in order to modify their own data variables. The variables can be used in the guards of the other processes, thus facilitating a natural communication among the processes.[1]

Figure 1 visualizes the intuition underlying environment abstraction. The grey box on the left hand side represents a concrete state of a system with 16 concurrent processes. The different colors of the disks/processes represent the internal states of the processes, i.e., the positions of the program counter.

The star-shaped graph on the right hand side of Figure 1 represents an abstract state. The abstract state contains one distinguished process – called the *reference process* x – which is at the center of the star. In this example, the reference process x represents process 1 of the concrete state. The disks on the circumference of the star represent the *environment* of the reference process. *Intuitively, the goal of the abstraction is to embed the reference process x of the abstract state into an abstract environment as rich as the environment which process 1 has in the concrete state.* Thus, the abstract state represents the concrete state "from the point of view of process 1."

[1] We assume that transitions involving global conditions are treated atomically, i.e., while a process is evaluating e.g. a guard, no other process makes any transition. This simplification – which we shall call the *atomicity assumption* further on – is implicit in other works on parameterized verification, see [3, 5, 6, 29].

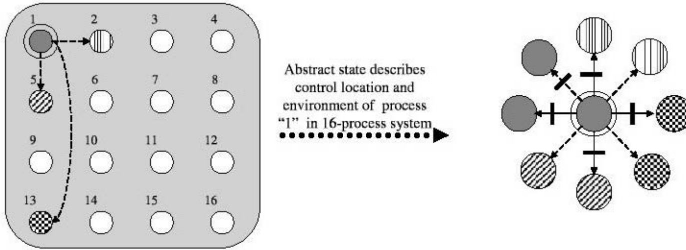


Fig. 1. Abstraction Mapping

To describe the environment of a process, we need to consider the relationships which can hold between the data variables of two processes. We can graphically indicate a specific relationship between any two processes by a corresponding arrow between the processes; the form of the arrow (full, dashed, etc.) determines *which* relationship the two processes have. In the figure, we assume that we have only two relationships R_1, R_2. For example, $R_1(x, y)$ might say that the local variable t of process x has the same value as local variable t in process y, while $R_2(x, y)$ might say that t has different values in processes x and y. Relationship R_1 is indicated by a full arrow, and R_2 is indicated by a dashed arrow. For better readability, not all relationships between the 16 processes are drawn.

More precisely, the environment of the reference process is described as follows: we enumerate all cases how the data variables in the reference process can relate to the data variables in a different process, as well as all possible program counter values which the other process can take. In our example, we have 2 relationships R_1, R_2 and 4 program counter positions, giving 8 different *environment conditions*. Therefore, the abstract state contains 8 environment processes on the circumference. For each of these 8 environment conditions, we indicate by the absence or presence of a *bar*, if this environment condition is actually satisfied by some process in the concrete state. For example, the dashed arrow from process 1 to the vertically striped process 2 in the concrete state necessitates a dashed arrow from x to a vertically striped process in the abstract state. Similarly, since there is no full arrow starting at process 1 in the concrete state, all full arrows in the abstract state have a bar. An *environment predicate* is a quantified formula which indicates the presence or absence of an environment condition for the reference process. We will give a formal definition of these notions in Section 4.

Note that a single abstract state in general represents an infinite number of concrete states. Moreover, a given concrete state gives rise to several abstract states, each of which is induced by choosing a different possible reference process. For example, the concrete state in Figure 1 may induce up to 16 abstract states, one for each process.

Existential Abstraction for Parameterized Systems. We construct an abstract system by a variant of existential abstraction. We include an abstract transition if *in some concrete instance of the parameterized system we can find a concrete transition between concrete states which match the abstract states with respect to the same reference process.* The abstract model obtained by environment abstraction is a sound abstraction which preserves both safety and liveness properties.

In this paper we use a simple input language which is general enough to describe most practically relevant symmetric protocols, and to demonstrate the underlying principles of our abstraction. We believe that our abstraction method can be naturally generalized for additional constructs as well.

To handle liveness we augment the abstract model using an approach suggested by [29]. Note that in contrast to the indexed predicates method [24, 25], our approach constructs an abstract transition system, instead of computing the set of reachable abstract states. This feature of our approach is crucial for verifying liveness properties.

Tool Chain and Experiments. Our approach provides an automated tool chain in the tradition of model checking.

1. The user feeds the protocol described in our language to the verification tool.
2. The environment abstraction tool extracts a finite state model from the process description, and puts the model in NuSMV format.
3. NuSMV verifies the specified properties.

Using the abstraction method described here, we have been able to verify automatically the safety and liveness properties of two well known mutual exclusion algorithms, namely Lamport's Bakery algorithm [26] and Szymanski's algorithm [31]. While safety and liveness properties of Szymanski's algorithm have been automatically verified with atomicity assumption by Baukus et al [5], this is the first time both safety and liveness of Lamport's bakery algorithm have been verified (with the atomicity assumption) at this level of automation.

2 Discussion of Related Work

Verification of parameterized systems is well known to be undecidable [2, 30]. Many interesting approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques [1, 23, 8, 7], invariant based techniques [3, 28], predicate abstraction [24], or exploiting symmetry [11, 14, 17, 15, 16]. Some of the earliest work on verifying parameterized systems includes works by Browne et al [9], German and Sistla [20], Emerson and Sistla [16]. In the rest of this section, we will concentrate on the work which is closest to our approach.

Counter Abstraction [4, 12, 13, 29, 20] is an intuitive method to use on parameterized systems. Pnueli et al [29] who coined the term counter abstraction show how systems composed of symmetric and finite state processes can be handled automatically. Protocols which either break symmetry by exploiting knowledge of process id's or which have infinite state spaces however require manual intervention. Thus, the verification of Szymanski's and the Bakery protocol in [29] requires manual introduction of new variables. The method also makes assumptions on the atomicity of guards.

The *Invisible Invariants* method was introduced in a series [28, 3, 18, 19] of papers. The idea behind this technique is to find an invariant for the parameterized system by examining concrete systems for low valuations of the parameter(s). The considered system model is powerful enough to model various mutual exclusion and cache coherence protocols which do not need unbounded integer variables. In [3], a modified version

of the bakery algorithm is verified: the original bakery algorithm is modified to eliminate unbounded integer variables. In contrast, the method proposed in the current paper can handle the original bakery protocol without such modifications. The authors of [3] implicitly make the assumption that guards are evaluated atomically.

The *Indexed Predicates* method [24, 25] is a new form of predicate abstraction for infinite state systems. This method relies on the observation that complex invariants are built of simple *indexed predicates*, i.e., predicates which have free index variables. By choosing a set of indexed predicates appropriately one can use a modified form of predicate abstraction to find a system invariant. In comparison to the above mentioned work, this method makes weaker atomicity assumptions.

Our method is also based on predicate abstraction; in fact, the notion of a reference process can be viewed as an "index" in the indexed predicates framework. However, the contribution we make is very different: (i) We focus on concurrent parameterized systems which enables us to use the specific and precise technique of environment abstraction. Our abstraction method exploits and reflects the structure of communicating protocols. (ii) In the indexed predicates approach there is no notion of an abstract transition relation. Thus, their approach, which is tailored for computing reachable states, works only for safety properties. In our framework, the abstract model does have a transition relation, and we can verify liveness properties as well as safety properties. (iii) The indexed predicate technique requires manual intervention or heuristics for choosing appropriate predicates. In contrast, our technique is automatic.

A method pioneered by Baukus et al [5] models an infinite class of systems by a single *WS1S system* which is then abstracted into a finite state system. While this is an automatic technique it cannot handle protocols such as the Bakery algorithm which have unbounded integer variables. The global conditions are assumed to be atomic.

The *inductive method* of [27] based on model checking is applied to verify both safety and liveness of the Bakery algorithm, notably without assuming atomicity. This approach however is not automatic: the user is required to provide lemmas and theorems to prove the properties under consideration. Our approach in contrast is fully automatic.

Regular model checking [8] is an interesting verification technique very different from ours. It is based on modeling systems using regular languages. This technique is applicable to a wide variety of systems but it requires the user to express systems in terms of regular languages which is a non-trivial process and requires user ingenuity.

Henzinger et. al. [22] also consider the problem of unbounded number of threads but the system model they consider is different. The communication between threads occurs through shared variables, whereas in our case, each process can look at the state of the other processes.

In summary, automatic methods such as the WS1S method and counter abstraction are restricted in the systems they can handle and make use of the atomicity assumption. In contrast, the methods which make no or weaker assumptions about atomicity tend to require user intervention, either in the form of providing appropriate predicates or in the form of lemmas and theorems which lead to the final result. In this paper, we assume atomicity of guards and describe a method which can handle well known mutual exclusion protocols such as the Bakery and Szymanski's protocols automatically. Importantly, our method is able to abstract and handle unbounded integer variables. To the

best of our knowledge, this is the first time that the Bakery algorithm (under atomicity assumption) has been verified automatically.

The method of environment abstraction described here has a natural extension which eliminates the atomicity assumption. This extension of our method, which will be described in future work, has been used to verify the Bakery algorithm and Szymanski's protocol without any restrictions.

3 System Model

Parameterized Systems. We consider asynchronous systems composed of an unbounded number of processes which communicate via shared variables. Each process can modify its own variables, but has only read access to the variables of the other processes. Each process has two sets of variables: the control variables $\mathbf{F} = \{f_1, \ldots, f_c\}$, where each f_i has a finite, constant range and the data variables $\mathbf{U} = \{u_1, \ldots u_d\}$, where each u_i is an unbounded integer. Intuitively, the two sets of variables serve different purposes: (i) The control variables in $\mathbf{F}$ determine the internal control state of the process. As they have a finite domain, the variables in $\mathbf{F}$ amount to the finite control of the process. (ii) The data variables in $\mathbf{U}$ contain actual data which can be read by other processes to calculate their own data variables.

All processes run the same protocol P. For a given protocol P, a system consisting of K processes running P will be denoted by $\mathcal{P}(K)$. Thus, the number K of processes is the system parameter. We will write $\mathcal{P}(\mathbf{N})$ to denote the infinite collection $\mathcal{P}(2), \mathcal{P}(3), \ldots$ of systems. To be able to refer to the state of individual processes in a system $\mathcal{P}(K)$ we will assume that each process has a distinct and fixed *process id* from the range $[1..K]$. We will usually refer to processes and their variables via their process id's. In particular, $f_a[i]$ and $u_b[i]$ denote the variables f_a and u_b of the process with id i. The set of *local states* of a process i is then naturally given by the different valuations of the tuple $\langle f_1[i], \ldots, f_c[i], u_1[i], \ldots, u_d[i]\rangle$. The global state of system $\mathcal{P}$(K) is given by a tuple $\langle \mathcal{L}_1, \ldots, \mathcal{L}_K\rangle$, where each $\mathcal{L}_i$ is the local state of process i. The initial state of each process is given by a fixed valuation of the local state variables. Note that all processes in a system $\mathcal{P}(K)$ are identical except for their $id's$. Thus, the process id's are the only means to break the symmetry between the processes. A process can use the reserved expression slf to refer to its own process id. When a protocol text contains the variables f_a or u_b without explicit reference to a process id, then this stands for $f_a[\mathsf{slf}]$ and $u_b[\mathsf{slf}]$ respectively.

A concrete valuation of the variables in $\mathbf{F}$ determines the control state of a process. *Without loss of generality, we can assume for simplicity that* $\mathbf{F}$ *has only one variable* pc *which determines the control state of a process.* Thus, in the rest of the paper $\mathbf{F} = \{\mathsf{pc}\}$, although in program texts we may take the freedom to use more than one finite range control variable. A formula of the form $\mathsf{pc} = \mathrm{const}$ is called a *control assignment*. The range of pc is called the set of control locations.

Guarded Transitions and Update Transitions. We will describe the transition relation of the processes in terms of two basic constructs, *guarded transitions* for the finite control, and the more complicated *update transitions* for modifying data variables. A guarded transition has the form

$$\mathsf{pc} = L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } \mathsf{pc} = L_2 \textbf{ else goto } \mathsf{pc} = L_3$$

or shorter

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3$$

where L_1, L_2, L_3 are control locations. In the guard $\forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ the variable otr ranges over the process id's of all other processes. The condition $\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ is any formula involving the data variables of processes slf, otr and the pc variable of otr. The semantics of a guarded transition is straightforward: in control location L_1, the process evaluates the guard and changes to control location L_2 or L_3 accordingly.

Update transitions are needed to describe protocols such as the Bakery algorithm where a process *computes* a data value depending on all values which it can read from other processes. For example, the Bakery algorithm has to compute the maximum of a certain data variable (the "ticket variable") in all other processes. Thus, we define an update transition to have the general form

$$\begin{aligned} L_1 : \quad & \textbf{for all } \mathsf{otr} \neq \mathsf{slf} \quad \textbf{if } \mathcal{T}(\mathsf{slf}, \mathsf{otr}) \textbf{ then } u_k := \phi(\mathsf{otr}) \\ & \textbf{goto } L_2 \end{aligned}$$

where L_1 and L_2 are control assignments, and $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is a condition involving data variables of processes $\mathsf{slf}, \mathsf{otr}$. The semantics of the update transition is best understood in an operational manner: In control location L_1, the process scans over all the other processes (in nondeterministically chosen order), and for each process otr checks if the formula $\mathcal{T}(\mathsf{slf}, \mathsf{otr})$ is true. In this case, the process changes the value of its data variable u_k according to $u_k := \phi(\mathsf{otr})$, where $\phi(\mathsf{otr})$ is an expression involving variables of process otr. Thus, the variable u_k can be reassigned multiple times within a transition. Finally, the process changes to control location L_2. *We assume that both guarded and update transitions are atomic, i.e., during their execution no other process makes a move.*

Example 1. As an example of a protocol written in this language, consider a parameterized system $\mathcal{P}(\mathbf{N})$ where each process P has one finite variable $\mathsf{pc} : \{1, 2, 3\}$ representing a program counter, one unbounded/integer variable $t : \mathbf{Int}$, and executes the following program:

$$\begin{aligned} &1 : \ \textbf{goto } 2 \\ &2 : \ \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.t[\mathsf{slf}] \neq t[\mathsf{otr}] \textbf{ then goto } 3 \\ &3 : \ t := t[\mathsf{otr}] + 1; \quad \textbf{goto } 1 \end{aligned}$$

The statement $1 : \ \textbf{goto} \ 2$ is syntactic sugar for

$$\mathsf{pc} = 1 : \ \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\textbf{true then goto } \mathsf{pc} = 2 \textbf{ else goto } 1$$

Similarly, $3 : \ t := t[\mathsf{otr}] + 1; \quad \textbf{goto} \ = 1$ is syntactic sugar for

$$\mathsf{pc} = 3 : \ \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\textbf{true then } t := t[\mathsf{otr}] + 1 \textbf{ goto } \mathsf{pc} = 1.$$

This example also illustrates that most commonly occurring transition statements in protocols can be written in our input language. □

Note that we have not specified the operations and predicates which are used in the conditions and assignments. Essentially, this choice depends on the protocols and the power of the decision procedures used. For the protocols considered in this paper, we need linear order and equality on data variables as well as incrementation, i.e., addition by 1. The full version of the paper [10] contains the descriptions of the Bakery algorithm and Szymanski's algorithm in terms of our language.

4 Environment Abstraction

In this section, we describe the principal framework of environment abstraction. In Section 5 we will discuss how to actually compute abstract models for the class of parameterized systems introduced in the previous section. Both tasks are non-trivial, as we need to construct a finite abstract model which reflects the properties of $\mathcal{P}(K)$ for all $K \geq 1$. We shall write $\mathcal{P}(\mathbf{N}) \models \Phi$ to say that $\mathcal{P}(K) \models \Phi$ for all parameters $K > 1$. Given a specification Φ and a system $\mathcal{P}(\mathbf{N})$, we will construct an abstract model $\mathcal{P}^{\mathcal{A}}$ and an abstract specification $\Phi^{\mathcal{A}}$ such that $\mathcal{P}^{\mathcal{A}} \models \Phi^{\mathcal{A}}$ implies $\mathcal{P}(\mathbf{N}) \models \Phi$. The converse does not have to hold, i.e., the abstraction is sound but not complete.

We will first describe how to construct the abstract model. We have already informally visualized and discussed the abstraction concept using Figure 1. More formally, our approach is best understood by viewing the abstract state as a *description* $\Delta(x)$ of the computing environment of a reference process x. Since x is a variable, we can then meaningfully say that the description $\Delta(x)$ holds true or false for a concrete process. We write $g \models \Delta(p)$ to express that in a global state g, $\Delta(x)$ holds true for the process p.

An abstract state (i.e., a description $\Delta(x)$) contains (i) detailed information about the current internal state of x and (ii) information about the internal states of other processes and their relationship to x. Since the number of other processes is not fixed, we can either *count* the number of processes which are in a given relationship to x, or, as in the current paper, keep track of the *existence* of such processes.

Technically, our descriptions reuse the predicates which occur in the control statements of the protocol description. Let S be the number of control locations in the program P. The internal state of a process x can be described by a predicate of the form

$$pc[x] = L$$

where $L \in \{1..S\}$ is a control location.

In order to describe the relations between the data variables of different processes we collect *all* predicates $\mathcal{EP}_1(x, y), \ldots, \mathcal{EP}_r(x, y)$ which occur in the guards of the program. From now on we will refer to these predicates as the *inter-predicates* of the program. Since in most practical protocols, synchronization between processes involves only one or two data variables, the number of inter-predicates is usually quite small. The relationship between a process x and a process y is now described by a formula of the form

$$R_i(x, y) \doteq \pm\mathcal{EP}_1(x, y) \wedge \ldots \wedge \pm\mathcal{EP}_r(x, y)$$

where $\pm\mathcal{EP}_i$ stands for $\mathcal{EP}_i$ or its negation $\neg\mathcal{EP}_i$. It is easy to see that there are 2^r possible relationships $R_1(x, y), \ldots, R_{2^r}(x, y)$ between x and y. In the example of Figure 1, the two relationship predicates R_1, R_2 are visualized by full and dashed arrows.

Fact 1. *The relationship conditions $R_1(x,y),\ldots,R_{2^r}(x,y)$ are mutually exclusive.*

Before we explain the descriptions $\Delta(x)$ in detail, let us first describe the most important building blocks for the descriptions which we call *environment predicates*. An environment predicate expresses that for process x we can find another process y which has a given relationship to process x and a certain internal state. The environment predicates thus have the form

$$\exists y.y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = j.$$

An environment predicate says the following: *there exists a process y different from x whose relationship to x is described by the $\mathcal{EP}$ predicates in R_i, and whose internal state is j.* There are $T := 2^r \times S$ different environment predicates; we name them $\mathcal{E}_1(x),\ldots,\mathcal{E}_T(x)$, and their quantifier-free matrices $E_1(x,y),\ldots,E_T(x,y)$. Note that each $E_k(x,y)$ has the form $y \neq x \wedge R_i(x,y) \wedge \mathsf{pc}[y] = j$.

Fact 2. *If an environment process y satisfies an environment condition $E_i(x,y)$, then it cannot simultaneously satisfy any other environment condition $E_j(x,y)$, $i \neq j$.*

Fact 3. *Let $E_i(x,y)$ be an environment condition and $\mathcal{G}(x,y)$ be a boolean formula over the inter-predicates $\mathcal{EP}_1(x,y),\ldots,\mathcal{EP}_r(x,y)$ and predicates of the form $\mathsf{pc}[y] = L$. Then either $E_i(x,y) \Rightarrow \mathcal{G}(x,y)$ or $E_i(x,y) \Rightarrow \neg\mathcal{G}(x,y)$.*

We are ready to return to the descriptions $\Delta(x)$. A description $\Delta(x)$ has the format

$$\mathsf{pc}[x] = i \quad \wedge \quad \pm\mathcal{E}_1(x) \wedge \pm\mathcal{E}_2(x) \wedge \cdots \wedge \pm\mathcal{E}_T(x), \quad \text{where } i \in [1..S]. \qquad (*)$$

Intuitively, a description $\Delta(x)$ therefore gives detailed information on the internal state of process x, and how the other processes are related to process x. Note the correspondence of $\Delta(x)$ to the abstract state in Figure 1: the control location i determines the color of the central circle, and the $\mathcal{E}_j$ determine the processes surrounding the central one.

We will now represent descriptions $\Delta(x)$ by tuples of values, as usual in predicate abstraction. The possible descriptions $(*)$ only differ in the value of the program counter $\mathsf{pc}[x]$ and in where they have negations in front of the $\mathcal{E}$ predicates. Denoting negation by 0 and absence of negation by 1, every description $\Delta(x)$ can be identified with a tuple $\langle \textbf{pc}, e_1, \ldots e_T\rangle$ where **pc** is a control location, and each e_i is a boolean variable. From this point of view, we have two ways to speak about abstract states: as descriptions $\Delta(x)$, and as tuples $\langle \textbf{pc}, e_1,\ldots,e_T\rangle$. Thinking of abstract states as descriptions is more intuitive in the conceptual phase of this work, while the latter approach is more in line with traditional predicate abstraction, and closer to the algorithms we use.

Example 2. Consider again the protocol shown in Example 1. There is only one inter-predicate $\mathcal{EP}_1(x,y) \doteq t[x] \neq t[y]$. Thus we have two possible relationship conditions $R_1(x,y) \doteq t[x] = t[y]$ and $R_2(x,y) \doteq t[x] \neq t[y]$. Consequently, we have 6 different environment predicates:

$$\begin{array}{ll}
\mathcal{E}_1(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_1(x,y) & \mathcal{E}_4(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 1 \wedge R_2(x,y) \\
\mathcal{E}_2(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_1(x,y) & \mathcal{E}_5(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 2 \wedge R_2(x,y) \\
\mathcal{E}_3(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_1(x,y) & \mathcal{E}_6(x) \doteq \exists y \neq x.\mathsf{pc}[y] = 3 \wedge R_2(x,y)
\end{array}$$

The abstract state then is a 7-tuple $\langle \textbf{pc}, e_1, \ldots, e_6 \rangle$ where **pc** refers to the internal state of the reference process x. For each $i \in [1..6]$, the bit e_i tells whether there is an environment process $y \neq x$ such that the environment predicate $\mathcal{E}_i(x)$ becomes true. □

Definition 1 (Abstract States). *Given a parameterized system $\mathcal{P}(\mathbf{N})$ with control locations $\{1, .., S\}$ and environment predicates $\mathcal{E}_1(x), \ldots, \mathcal{E}_T(x)$, the abstract state space contains tuples $\langle \textbf{pc}, e_1, \ldots e_T \rangle$, where*

- $\textbf{pc} \in \{1, .., S\}$ *denotes the control location of the reference process.*
- *each e_j is a Boolean variable corresponding to the predicate $\mathcal{E}_j(x)$.*

Since the concrete system $\mathcal{P}(K)$ contains K processes, a state $s \in \mathcal{P}(K)$ can give rise to up to K different abstract states, one for every different choice of the reference process.

Definition 2 (Abstraction Mapping). *Let $P(K)$, $K > 1$, be a concrete system and $p \in [1..K]$ be a process. The abstraction mapping α_p induced by p* maps *a global state g of $\mathcal{P}(K)$ to an abstract state $\langle \textbf{pc}, e_1, \ldots, e_T \rangle$ where*

$$\textbf{pc} = \textit{ the value of } \mathsf{pc}[p] \textit{ in state } g \quad \textit{and for all } e_j \textit{ we have } e_j = 1 \Leftrightarrow g \models \mathcal{E}_j(p).$$

Definition 3 (Abstract Model). *The abstract model $\mathcal{P}^{\mathcal{A}}$ is given by the transition system $(S^{\mathcal{A}}, \Theta^{\mathcal{A}}, \rho^{\mathcal{A}})$ where*

- $S^{\mathcal{A}} = \{1, .., S\} \times \{0, 1\}^T$*, the set of abstract states, contains all valuations of the tuple $\langle \textbf{pc}, e_1, \ldots, e_T \rangle$.*
- $\Theta^{\mathcal{A}}$*, the set of initial abstract states, is the set of abstract states $\hat{s}$ such that there exists a concrete initial state s of a concrete system $\mathcal{P}(K)$, $K > 1$, such that there exists a concrete process p with $\alpha_p(s) = \hat{s}$.*
- $\rho^{\mathcal{A}} \subseteq S^{\mathcal{A}} \times S^{\mathcal{A}}$ *is a transition relation on the abstract states defined as follows: There is a transition from abstract state $\hat{s}_1$ to abstract state $\hat{s}_2$ if* there exist
  (i) a concrete system $\mathcal{P}(K), K > 1$ with a process p
  (ii) a concrete transition from concrete state s_1 to s_2 in $\mathcal{P}(K)$

  such that $\alpha_p(s_1) = \hat{s}_1$ *and* $\alpha_p(s_2) = \hat{s}_2$.

4.1 Specifications

We will now focus on the properties that we want to verify. By a *one process* control condition we mean a boolean formula over expressions of the form $\mathsf{pc}[x] = L, L \in \{1, .., S\}$. By a two process control condition we mean a boolean formula over expressions of the form $\mathsf{pc}[x] = L_1, \mathsf{pc}[y] = L_2,$ where $L_1, L_2 \in \{1, .., S\}$.

Definition 4 (Two-Indexed Safety Properties). *A two-indexed safety property is a specification $\forall x, y.\ \mathbf{AG}\phi(x, y)$, where x, y are variables which refer to* distinct *processes, and $\phi(x, y)$ is a two process control condition.*

Definition 5 (Liveness Properties). *A liveness property is a specification of the form $\forall x.\ \mathbf{AG}(\phi(x) \rightarrow \mathbf{F}\psi(x))$, where $\phi(x)$ and $\psi(x)$ are one process control conditions.*

A standard example of a two-indexed safety property is the mutual exclusion property $\forall x, y.\ \mathbf{AG}\neg(\mathsf{pc}[x] = \text{crit} \wedge \mathsf{pc}[y] = \text{crit})$, where crit is the control location of the critical section. An example of a liveness property is the formula $\forall x.\ \mathbf{AG}\ (\mathsf{pc}[x] = \text{try} \rightarrow \mathbf{F}\ \mathsf{pc}[x] = \text{crit})$ which expresses that a process gets to enter the critical section if it wants to.

We first show how to abstract a formula $\phi(x, y)$ without any temporal operators. The abstraction $\phi^{\mathcal{A}}$ of $\phi(x, y)$ is a predicate over the abstract states that is satisfied by those and only those abstract states $\hat{s}$ for which there exists a system $\mathcal{P}(K)$, $K > 1$ with a process p, and a global state s of $\mathcal{P}(K)$ such that

$$\alpha_p(s) = \hat{s} \text{ and } \forall q \neq p.\ (s \models \phi(p, q)).$$

Intuitively, we treat x as the reference process and y as an environment process and find which abstract states correspond to the concrete formula $\phi(x, y)$. Similarly, for a single index property $\phi(x)$, its abstraction $\phi^{\mathcal{A}}$ is the predicate that is satisfied by those and only those abstract states $\hat{s}$ for which there exists a system $\mathcal{P}(K)$, $K > 1$, with a process p and a global state s of $\mathcal{P}(K)$ such that $\alpha_p(s) = \hat{s}$ and $s \models \phi(p)$.

Now we can define the abstract specifications: The abstraction of a two-indexed safety property $\forall x, y.\ \mathbf{AG}\phi(x, y)$ is the formula $\mathbf{AG}\phi^{A}$. The abstraction of a single-indexed liveness property $\forall x.\ \mathbf{AG}\ (\phi(x) \rightarrow \mathbf{F}\psi(x))$ is the formula $\mathbf{AG}\ (\phi^{A} \rightarrow \mathbf{F}\psi^{A})$.

Theorem 1 (Soundness of Abstraction). *Let $\mathcal{P}(\mathbf{N})$ be a parameterized system and $\mathcal{P}^{\mathscr{A}}$ be an* over-approximation *of its abstraction $\mathcal{P}^{\mathcal{A}}$. Given any two-indexed safety or single-indexed liveness property Φ and its abstraction $\Phi^{\mathcal{A}}$ we have $\mathcal{P}^{\mathscr{A}} \models \Phi^{\mathcal{A}}$ implies $\mathcal{P}(\mathbf{N}) \models \Phi$.*

4.2 Extensions for Fairness and Liveness

The abstract model that we have described, while sound, might be too coarse in practice to be able to verify liveness properties. The reason is two fold:

(i) Spurious Infinite Paths. Our abstract model may have infinite paths which cannot occur in any concrete system. This happens when two concrete states s_1 and s_2, where s_1 transitions to s_2, both map to the same abstract state $\hat{s}$, leading to a self-loop involving $\hat{s}$. Such a self-loop can lead to a spurious infinite path which hinders the verification of liveness properties.

(ii) Fairness Conditions. Liveness properties are usually expected to hold under some fairness conditions. A typical example of a fairness condition is that every process x must leave the critical section a finite time after entering it. This is expressed formally by the fairness condition $\mathsf{pc}[x] \neq \text{crit}$. In this paper we will consider fairness conditions $\mathsf{pc}[x] \neq L$, where L is a control location. Liveness properties are then expected to hold on *fair paths*: an infinite path in a concrete system $\mathcal{P}(K)$, $K \geq 1$ is *fair* only if the fairness condition $\mathsf{pc}[i] \neq L$ holds for each process i infinitely often.

To handle these situations, we adapt a method developed by Pnueli et al. [29] in the context of counter abstraction to our environment abstraction. To this end, we augment our abstract model by adding new *Boolean* variables $\mathbf{from}_i, \mathbf{to}_i$ for every $i \in [1..T]$. Thus

our new abstract states are tuples $\langle \textsf{pc}, e_1, \ldots, e_T, \textsf{from}_1, \ldots, \textsf{from}_T, \textsf{to}_1, \ldots, \textsf{to}_T \rangle$. We will now briefly describe this extension.

Intuitively, the new **from**, **to** variables keep track of the *immediate* history of an abstract state, that is, the last step by which the abstract state was reached. The variable $\textsf{from}_i$ is **true** if a process y having satisfied $E_i(x, y)$ in the previous state does not satisfy $E_i(x, y)$ in the new state. Similarly, the variable $\textsf{to}_i$ is **true** if the active process having satisfied $E_j(x, y)$, $j \neq i$ in the previous state satisfies $E_i(x, y)$ in the new state. To eliminate the spurious infinite paths arising from loops described in item **(i)** above, we add for each $i \in [1..T]$ a *compassion condition* [29] $\langle \textsf{from}_i, \textsf{to}_i \rangle$ which says *If* $\textsf{from}_i = \mathbf{true}$ *holds infinitely often in a path, then* $\textsf{to}_i = \mathbf{true}$ *must hold infinitely often as well.*

Let us now turn to item **(ii)**. Given a concrete fairness condition of the form $\mathsf{pc}[x] \neq L$, the corresponding abstract fairness condition for the *reference process* is given by $\textsf{pc} \neq L$. Moreover, we introduce fairness conditions $\neg(\textsf{from}_i = \mathbf{false} \wedge e_i = 1)$ for all those environments $E_i(x, y)$ *which require process y to be in control location L*, i.e., those $E_i(x, y)$ which contain the subformula $\mathsf{pc}[y] = L$. For such an environment condition E_i, the fairness condition $\neg(\textsf{from}_i = \mathbf{false} \wedge e_i = 1)$ excludes the case that there are environment processes satisfying $E_i(x, y)$ which never move. For a more detailed explanation and proofs please consult the full version.

5 Computing the Abstract Model

In our implementation, we consider protocols in which all inter-predicates $\mathcal{EP}_i(x, y)$ have the form $t[x] \prec t[y]$ where $\prec \in \{<, >, =\}$ and t is a data variable.[2] Thus, each local process compares its own variables only with their counterparts in other processes. Most real protocols satisfy this condition. Our results however do not depend on this particular choice of inter-predicates.

Computing the abstract transition relation is evidently complicated by the fact that there is an infinite number of concrete systems. To get around this problem, we consider each concrete transition statement of the program separately and *over-approximate* the set of abstract transitions it can lead to. Their union will be our abstract transition relation.

A concrete transition can either be a guarded transition or an update transition. Each transition can be executed by the reference process or one of the environment processes. Thus there are four cases to consider:

Active process is ...	guarded transition	update transition
...*reference process*	Case 1	Case 2
...*environment process*	Case 3	Case 4

In this section we will consider Case 1, that is, the reference process executing the guarded transition. Computing the abstract transition in other cases is similar in spirit but quite lengthy. We refer the reader to the full version [10] of this paper for a more

[2] The incrementation operation occurs only on the right hand side of assignments in update transitions.

detailed description of how we compute the abstract initial condition and the abstract transition relation.

Let us now turn to Case 1 in detail, and consider the guarded transition

$$L_1 : \quad \textbf{if } \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr}) \textbf{ then goto } L_2 \textbf{ else goto } L_3. \qquad (*)$$

Suppose the reference process is executing this guarded transition statement. If at least one of the environment processes contradicts the guard $\mathcal{G}$ then the reference process transitions to control location L_3, i.e., the *else branch*. Otherwise, the reference process goes to L_2. We will now formalize the conditions under which the *if* and *else* branches are taken.

Definition 6 (Blocking Set for Reference Process). *Let $\mathscr{G} \doteq \forall \mathsf{otr} \neq \mathsf{slf}.\mathcal{G}(\mathsf{slf}, \mathsf{otr})$ be a guard. We say that an environment condition $E_i(x, y)$ blocks the guard $\mathscr{G}$ if $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$. The set $\mathcal{B}^x(\mathscr{G})$ of all indices i such that $E_i(x, y)$ blocks $\mathscr{G}$ is called the blocking set of the reference process for guard $\mathscr{G}$.*

Note that by Fact 3, either $E_i(x, y) \Rightarrow \neg\mathcal{G}(x, y)$ or $E_i(x, y) \Rightarrow \mathcal{G}(x, y)$ for every environment $E_i(x, y)$. The intuitive idea behind the definition is that $\mathcal{B}^x(\mathscr{G})$ contains the indices of all environment conditions which enforce the *else branch*. We will now explain how to represent the guarded transition $(*)$ in the abstract model: we introduce an abstract transition from $\hat{s_1} = \langle \mathbf{pc}, e_1, .., e_T, \mathbf{from}_1, .., \mathbf{from}_T, \mathbf{to}_1, .., \mathbf{to}_T \rangle$ to $\hat{s_2} = \langle \mathbf{pc}', e_1, .., e_T, \mathbf{from}'_1, .., \mathbf{from}'_T, \mathbf{to}'_1, .., \mathbf{to}'_T \rangle$ if

1. $\mathbf{pc} = L_1$, i.e., the reference process is in location L_1,
2. one of the following two conditions holds:
   - *If Branch:* $\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc}' = L_2$, i.e., the guard is true and the reference process moves to control state L_2.
   - *Else Branch:* $\neg\forall i \in \mathcal{B}^x(\mathscr{G}).\ (e_i = 0)$ and $\mathbf{pc}' = L_3$, i.e., the guard is false and the reference process moves to control state L_3.
3. all the variables $\mathbf{from}'_1, .., \mathbf{from}'_T$ and $\mathbf{to}'_1, .., \mathbf{to}'_T$ are **false**, expressing that none of the environment processes changes its state.

Thus, in order to compute the abstract transition we just need to find the blocking set $\mathcal{B}^x(\mathscr{G})$. This task is easy for predicates involving only linear order.

6 Experimental Results

We have implemented a prototype of our abstraction method in JAVA. As argued above, our implementation handles protocols in which all the predicates appearing in the guards involve only $\{<, >, =\}$. Thus, in this preliminary implementation, the decision problems that arise during the abstraction are simple and are handled by our abstraction program internally. We verified the safety and liveness properties of Szymanski's mutual exclusion protocol and Lamport's bakery algorithm. These two protocols have an intricate combinatorial structure and have been used widely as benchmarks for parameterized verification. For safety properties, we verified that no two processes can be

	Inter-preds	Intra-preds	Reachable states	Safety	Liveness
Szymanski	1	8	$O(2^{14})$	0.1s	1.82s
Bakery	3	5	$O(2^{146})$	68.55s	755.0s

Fig. 2. Running Times

present in the critical section at the same time. For liveness, we verified the property that if a process wishes to enter the critical section then it eventually will.

We used the NuSMV model checker to verify the finite abstract model. The model checking times are shown in Figure 2. The abstraction time is negligible, less than 0.1s. Figure 2 also shows the number of predicates and the size of the reachable state space as reported by NuSMV. All experiments were run on a 2.4 GHz Pentium machine with 512 MB main memory.

7 Conclusion

We have enriched predicate abstraction by ideas from counter abstraction to develop a new framework for verifying parameterized systems. We have applied this method to verify, under the atomicity assumption, the safety and liveness properties of two well known mutual exclusion protocols.

The main focus of this paper was the verification of small but very intricate systems. In these systems, the challenge is to handle the tightly inter-twined execution of an unbounded number of processes and to maintain predicates which are spanning multiple processes.

At the heart of our approach lies a notion of abstraction – *environment abstraction* – which describes the status of a concurrent system from the point of view of a single process. In addition to safety properties, environment abstraction naturally allows to verify fairness properties as well. The framework presented in this paper is a specific instance of environment abstraction tailored for distributed mutual exclusion algorithms. The general approach can be naturally extended in several ways:

- In this paper, the internal state of a process is described by a control location $\mathsf{pc} = L$. In a more general framework, the state of a process can be described using additional predicates which relate the different data variables of one process. This extension is quite straightforward but omitted from the current paper for the sake of simplicity.
- We have also extended the method to deal with systems in which there is a central process in addition to the K local processes. This extension allows us to handle *directory based cache coherence protocols* and will be reported in future work.
- The most important improvement of our results concerns the *elimination of the atomicity assumption* as to achieve automated protocol verification in a non-simplified setting for the first time. We recently have reached this goal by an extension of environment abstraction. We will report these results in future work.

To conclude, we want to emphasize that viewing a concurrent system from the point of view of a single process closely matches the reasoning involved in designing a dis-

tributed algorithm. We therefore believe that environment abstraction naturally yields powerful system abstractions.

Acknowledgments

The authors are grateful to the anonymous referees and Ken McMillan, and Lenore Zuck for discussions and comments which helped to improve the presentation of this paper.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model-checking made simple and efficient. In *Proc. 13th International Conference on Concurrency Theory (CONCUR)*, 2002.
2. K. Apt and D. Kozen. Limits for automatic verification of finite state concurrent systems. *Information Processing Letters*, 15:307–309, 1986.
3. T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13th Intl. Conf. Computer Aided Verification (CAV)*, 2001.
4. T. Ball, S. Chaki, and S. Rajamani. Verification of multi-threaded software libraries. In *ICSE*, 2001.
5. K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proc. TACAS*, 2000.
6. K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized protocols. In *Journal of Universal of Computer Science*, 2001.
7. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *15th Intern. Conf. on Computer Aided Verification (CAV'03)*. LNCS, Springer-Verlag, 2003.
8. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th Intern. Conf. on Computer Aided Verification (CAV'00)*. LNCS, Springer-Verlag, 2000.
9. M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.
10. E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *www.cs.cmu.edu/∼tmurali/vmcai06.ps*.
11. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal model checking. In *Proc. 5th Intl. Conf. Computer Aided Verification (CAV)*, 1993.
12. G. Delzanno. Automated verification of cache coherence protocols. In *Computer Aided Verification 2000 (CAV 00)*, 2000.
13. A. E. Emerson and V. Kahlon. Model checking guarded protocols. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 361–370, 2003.
14. E. A. Emerson, J. Havlicek, and R. Trefler. Virtual symmetry. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.
15. E. A. Emerson and A. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata theoretic approach. *TOPLAS*, 4, 1997.
16. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Intl. Conf. Computer Aided Verification (CAV)*, 1993.
17. E. A. Emerson and R. Trefler. From asymmetry to full symmetry. In *CHARME*, 1999.
18. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with incomprehensible ranking. In *Proc. VMCAI*, 2004.

19. Y. Fang, N. Piterman, A. Pnueli, and L. Zuck. Liveness with invisible ranking. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.
20. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39, 1992.
21. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. CAV 97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
22. T. Henzinger, R. Jhala, and R. Majumdar. Race checking with context inference. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2004.
23. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. CAV'97*, volume 1254 of *LNCS*, pages 424–435. Springer, June 1997.
24. S. K. Lahiri and R. Bryant. Constructing quantified invariants. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004.
25. S. K. Lahiri and R. Bryant. Indexed predicate discovery for unbounded system verification. In *Proc. 16th Intl. Conf. Computer Aided Verification (CAV)*, 2004.
26. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
27. K. L. McMillan, S. Qadeer, and J. B. Saxe. Induction in compositional model checking. In *Conference on Computer Aided Verfication*, pages 312–327, 2000.
28. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2001.
29. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$ counter abstraction. In *Computer Aided Verification 2002 (CAV 02)*, 2002.
30. I. Suzuki. Proving properties of a ring of finite state machines. *Information Processing Letters*, 28:213–214, 1988.
31. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc International Conference on Supercomputing Systems*, 1988.

Error Control for Probabilistic Model Checking*

Håkan L.S. Younes

Computer Science Department, Carnegie Mellon University,
Pittsburgh, PA 15213, USA

Abstract. We introduce a framework for expressing correctness guarantees of model-checking algorithms. The framework allows us to qualitatively compare different solution techniques for probabilistic model checking, both techniques based on statistical sampling and numerical computation of probability estimates. We provide several new insights into the relative merits of the different approaches. In addition, we present a new statistical solution method that can bound the probability of error under any circumstances by sometimes reporting undecided results. Previous statistical solution methods could only bound the probability of error outside of an "indifference region."

1 Introduction

Probabilistic model checking, based on the model-checking paradigm pioneered by Clarke and Emerson [4], is a technique for automated verification of stochastic processes. Given a model of a stochastic process, for example a Markov chain, the model-checking task is to determine whether the model satisfies some property Φ. For instance, consider a queuing system with random (according to some distribution) arrivals and departures. We may ask whether the probability is at most 0.5 that the queue will become full in the next hour of operation. This is an example of a probabilistic *time-bounded* property. Techniques for verifying such properties for stochastic discrete-event systems *without nondeterminism* are the focus of this paper.

Algorithms for probabilistic model checking of time-bounded properties come in two flavors: *numerical* [3, 12] and *statistical* [19, 8, 15, 17]. The former rely on numerical algorithms for probability computations, while the latter use statistical sampling and discrete-event simulation to assess the validity of probabilistic properties. Some insights into the relative merits of the two approaches are given by Younes et al. [18]. Yet, a direct comparison is difficult because numerical and statistical techniques provided quite different correctness guarantees. Furthermore, conflicting claims have been made about the benefits of competing statistical solution methods. Hérault et al. [8] state that their solution method, based on statistical *estimation*, is better than the method of Younes and Simmons [19], based on *hypothesis testing*, because the sample size of the former method is known exactly. Sen et al. [15] provide empirical data that seem to

* Supported in part by the US Army Research Office (ARO), under contract no. DAAD190110485.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 142–156, 2006.
© Springer-Verlag Berlin Heidelberg 2006

suggest that hypothesis testing with *fixed-size samples* consistently outperforms *sequential* hypothesis testing (the latter being advocated by Younes et al. [18]).

This paper is an attempt to set the record straight regarding the relative merits of different solution methods for probabilistic model checking. We establish a framework for expressing the correctness guarantees of model-checking algorithms (Sect. 3). Section 4 shows how to connect the truncation error, ϵ, of numerical methods with the parameter δ (the half-width of the "indifference region") of statistical methods. We conclude that numerical and statistical solution methods can, indeed, be interpreted as solving the same problem. Statistical solution methods are simply randomized algorithms for the same problems that numerical methods solve. We are also able to prove that statistical estimation, when used for probabilistic model checking, reduces to hypothesis testing with fixed-size samples. It follows that Younes and Simmons' solution method *never* needs to use a larger sample size than Hérault et al.'s estimation-based method, and it will often use a much smaller sample size to achieve the same correctness guarantees. Our framework for error control also helps us understand the results of Sen et al., which seem to contradict results presented by Younes [17].

The second contribution of this paper is a new statistical method for probabilistic model checking. Current statistical solution methods provide bounds for the probability of error only when a formula holds (or does not hold) *with some margin*. Our new method bounds the probability of error *under all circumstances*. This is accomplished by permitting an *undecided* result. The idea of undecided results for statistical solution methods is due to Sen et al. [15], but they did not provide any mechanisms for bounding the probability of producing an undecided result (or even an incorrect result, for that matter). Section 5 shows, for the first time, how to bound the probability of undecided and incorrect results for *any time-bounded formula*, including conjunctions of probabilistic statements and nested probabilistic statements. Section 6 discusses the computational complexity of statistical solution methods in general. A brief empirical evaluation of the new statistical solution method is provided in Sect. 7.

2 Probabilistic Model Checking

This section describes stochastic discrete-event systems (without nondeterminism), which is the class of models that we consider for probabilistic model checking. A logic, UTSL, for expressing properties of such models is introduced. We describe the semantics of UTSL and of UTSL_δ, the latter being a relaxation of the former logic that permits practical model-checking algorithms.

2.1 Stochastic Discrete-Event Systems

A *stochastic discrete-event system* is any stochastic process that can be thought of as occupying a single state for a duration of time before an *event* causes an instantaneous state transition to occur. The canonical example is a queuing system, with the state being the number of items currently in the queue. The state changes at the occurrence of an arrival or departure event.

The evolution of a stochastic discrete-event system over time is captured by a *trajectory*. The trajectory of a stochastic discrete-event system is piecewise constant and can be represented as a sequence $\sigma = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \ldots\}$, with $s_i \in S$ and $t_i > 0$. Let

$$T_i = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j=0}^{i-1} t_j & \text{if } i > 0 \end{cases} , \tag{1}$$

so that T_i is the time at which state s_i is entered and t_i is the duration of time for which the process remains in s_i before an event triggers a transition to state s_{i+1}. It is assumed that $\lim_{i\to\infty} T_i = \infty$. This implies that only a finite number of events can trigger in a finite interval of time, which is a reasonable assumption for any physical process (cf. [1]).

A *measurable* stochastic discrete-event system is a triple $\mathcal{M} = \langle S, T, \mu\rangle$, where S is the state space, T is the time domain ($\mathbb{Z}^*$ for discrete-time models and $[0, \infty)$ for continuous-time models), and μ is a probability measure over sets of trajectories with *common prefix*. A prefix of $\sigma = \{\langle s_0, t_0\rangle, \langle s_1, t_1\rangle, \ldots\}$ is a sequence $\sigma_{\ \tau} = \{\langle s'_0, t'_0\rangle, \ldots, \langle s'_k, t'_k\rangle\}$, with $s'_i = s_i$ for all $i \le k$, $\sum_{i=0}^{k} t'_i = \tau$, $t'_i = t_i$ for all $i < k$, and $t'_k < t_k$. Let $Path(\sigma_{\ \tau})$ denote the set of trajectories with common prefix $\sigma_{\ \tau}$. This set must be measurable for probabilistic model checking to have meaning and its measure is determined by μ. The exact definition of μ depends on the structure of the process. Baier et al. [3] provide a definition for continuous-time Markov chains and Younes [17] discusses the construction of a probability space for trajectories of stochastic discrete-event systems in general (see also, Segala's [14] definition of *trace distributions*).

2.2 UTSL: The Unified Temporal Stochastic Logic

A stochastic discrete-event system is a triple $\langle S, T, \mu\rangle$. We assume a factored representation of S, with a set of state variables SV and a value assignment function $V(s, x)$ providing the value of $x \in SV$ in state s. The domain of x is the set $D_x = \bigcup_{s\in S} V(s, x)$ of possible values that x can take on. We define the syntax of UTSL for a factored stochastic discrete-event system $\mathcal{M} = \langle S, T, \mu, SV, V\rangle$ as

$$\Phi ::= x \sim v \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{\bowtie\,\theta}[\Phi\ \mathcal{U}^I\ \Phi] ,$$

where $x \in SV$, $v \in D_x$, $\sim\, \in \{\le, =, \ge\}$, $\theta \in [0, 1]$, $\bowtie\, \in \{\le, \ge\}$, and $I \subset T$. Additional UTSL formulae can be derived in the usual way. For example, $\bot \equiv (x = v) \wedge \neg(x = v)$ for some $x \in SV$ and $v \in D_x$, $\top \equiv \neg\bot$, $\Phi \vee \Psi \equiv \neg(\neg\Phi \wedge \neg\Psi)$, $\Phi \to \Psi \equiv \neg\Phi \vee \Psi$, and $\mathcal{P}_{<\,\theta}[\varphi] \equiv \neg\mathcal{P}_{\ge\,\theta}[\varphi]$.

The standard logic operators have their usual meaning. $\mathcal{P}_{\bowtie\,\theta}[\varphi]$ asserts that the probability measure over the set of trajectories satisfying the path formula φ is related to θ according to $\bowtie$. Path formulae are constructed using the temporal path operator $\mathcal{U}^I$ ("until"). The path formula $\Phi\ \mathcal{U}^I\ \Psi$ asserts that Ψ becomes true $t \in I$ time units into the future while Φ holds continuously prior to t. The validity of a UTSL formula is inductively defined as follows:

$$
\begin{array}{rl}
\mathcal{M}, \{\langle s_0, t_0\rangle, \ldots, \langle s_k, t_k\rangle\} \models x \sim v & \text{if } V(s_k, x) \sim v \\
\mathcal{M}, \sigma\ _{\tau} \models \neg\Phi & \text{if } \mathcal{M}, \sigma\ _{\tau} \not\models \Phi \\
\mathcal{M}, \sigma\ _{\tau} \models \Phi \wedge \Psi & \text{if } (\mathcal{M}, \sigma\ _{\tau} \models \Phi) \wedge (\mathcal{M}, \sigma\ _{\tau} \models \Psi) \\
\mathcal{M}, \sigma\ _{\tau} \models \mathcal{P}_{\bowtie\theta}[\varphi] & \text{if } \mu(\{\sigma \in \mathit{Path}(\sigma\ _{\tau}) \mid \mathcal{M}, \sigma, \tau \models \varphi\}) \bowtie \theta
\end{array}
$$

$$
\begin{array}{rl}
\mathcal{M}, \sigma, \tau \models \Phi\, \mathcal{U}^{I}\ \Psi & \text{if } \exists t \in I.\big((\mathcal{M}, \sigma\ _{\tau+t} \models \Psi) \\
& \qquad \wedge \forall t' \in T.\big((t' < t) \rightarrow (\mathcal{M}, \sigma\ _{\tau+t}\ \models \Phi)\big)\big)
\end{array}
$$

The semantics of $\Phi\ \mathcal{U}^{I}\ \Psi$ requires that Φ holds continuously, i.e. at every point in time, along a trajectory until Ψ is satisfied. For Markov chains, it is sufficient to consider time points at which state transitions occur. The semantics of UTSL therefore coincides with the semantics for Hansson and Jonsson's [7] PCTL interpreted over discrete-time Markov chains and Baier et al.'s [3] CSL interpreted over continuous-time Markov chains. For non-Markovian models, however, the validity of Φ or Ψ may vary over time in the same state if these formulae contain probabilistic operators. Because of this, the statistical solution method for probabilistic model checking presented in this paper is restricted to Markov chains for properties with nested probabilistic operators. Without nesting, the method does not rely on this restriction.

We typically want to know whether a property Φ holds for a model $\mathcal{M}$ if execution starts in a specific state s. A *model-checking problem* $\langle \mathcal{M}, s, \Phi\rangle$ has an affirmative answer if and only if $\mathcal{M}, \{s, 0\} \models \Phi$.

2.3 UTSL$_\delta$: UTSL with Indifference Regions

Consider the model-checking problem $\langle \mathcal{M}, s, \mathcal{P}_{\bowtie\theta}[\varphi]\rangle$ and let p be the probability measure for the set of trajectories that start in s and satisfy φ. If p is "sufficiently close" to θ, then it is likely to make little difference to a user whether or not $\mathcal{P}_{\bowtie\theta}[\varphi]$ is reported to hold by a model-checking algorithm.

To formalize this idea, we introduce UTSL$_\delta$ as a relaxation of UTSL. With each formula of the form $\mathcal{P}_{\bowtie\theta}[\varphi]$, we associate an indifference region centered around θ with half-width δ. If $|p - \theta| < \delta$, then the truth value of $\mathcal{P}_{\bowtie\theta}[\varphi]$ is undetermined for UTSL$_\delta$; otherwise, it is the same as for UTSL.

The formal semantics of UTSL$_\delta$ is given by a satisfaction relation $|\!\!\approx^{\delta}$ and an unsatisfaction relation $|\!\!\approx^{\delta}$. For standard logic formulae, $|\!\!\approx^{\delta}$ replaces $\models$ and $|\!\!\approx^{\delta}$ replaces $\not\models$. For probabilistic formulae we have the following rules:

$$
\begin{array}{ll}
\mathcal{M}, \sigma\ _{\tau} |\!\!\approx^{\delta}\ \mathcal{P}_{\geq\theta}[\varphi] & \text{if } \mu(\{\sigma \in \mathit{Path}(\sigma\ _{\tau}) \mid \mathcal{M}, \sigma, \tau |\!\!\approx^{\delta}\ \varphi\}) \geq \theta + \delta \\
\mathcal{M}, \sigma\ _{\tau} |\!\!\approx^{\delta}\ \mathcal{P}_{\geq\theta}[\varphi] & \text{if } \mu(\{\sigma \in \mathit{Path}(\sigma\ _{\tau}) \mid \mathcal{M}, \sigma, \tau |\!\!\approx^{\delta}\ \varphi\}) \geq 1 - (\theta - \delta) \\
\mathcal{M}, \sigma\ _{\tau} |\!\!\approx^{\delta}\ \mathcal{P}_{\leq\theta}[\varphi] & \text{if } \mu(\{\sigma \in \mathit{Path}(\sigma\ _{\tau}) \mid \mathcal{M}, \sigma, \tau |\!\!\approx^{\delta}\ \varphi\}) \leq \theta - \delta \\
\mathcal{M}, \sigma\ _{\tau} |\!\!\approx^{\delta}\ \mathcal{P}_{\leq\theta}[\varphi] & \text{if } \mu(\{\sigma \in \mathit{Path}(\sigma\ _{\tau}) \mid \mathcal{M}, \sigma, \tau |\!\!\approx^{\delta}\ \varphi\}) \leq 1 - (\theta + \delta)
\end{array}
$$

A model-checking problem $\langle \mathcal{M}, s, \Phi\rangle$ may very well belong to neither of the two relations $|\!\!\approx^{\delta}$ and $|\!\!\approx^{\delta}$, in which case the problem is considered "too close to call."

3 Error Control

This section discusses error control for model-checking algorithms in general terms. The discussion establishes ideal conditions for the correctness guarantees of a model-checking algorithm. These conditions are used as a point of reference in later sections when we discuss error control in practical algorithms for probabilistic model checking.

Given a model-checking problem $\langle \mathcal{M}, s, \Phi \rangle$ and a model-checking algorithm $\mathcal{A}$, let $\mathcal{M}, s \vdash \quad \Phi$ represent the fact that Φ is accepted as true by $\mathcal{A}$ and $\mathcal{M}, s \vdash \quad \Phi$ that Φ is rejected as false by $\mathcal{A}$ (for the remainder of the paper we will leave out $\mathcal{M}$ from relations for the sake of brevity). Ideally, we would like the probability to be low that $\mathcal{A}$ produces an incorrect answer. More precisely, the probability of a false negative should be at most α and the probability of a false positive at most β, as expressed by the following conditions:

$$\Pr[s \vdash \quad \Phi \mid s \models \Phi] \leq \alpha \tag{2}$$

$$\Pr[s \vdash \quad \Phi \mid s \not\models \Phi] \leq \beta \tag{3}$$

In addition, the probability should be low that $\mathcal{A}$ does not produce a definite answer. Let $s \vdash \quad \Phi$ denote that $\mathcal{A}$ is *undecided.* We add

$$\Pr[s \vdash \quad \Phi] \leq \gamma \tag{4}$$

to represent this requirement. Finally, $\mathcal{A}$ should always terminate with one of the three possible answers (accept, reject, or undecided):

$$\Pr[(s \vdash \quad \Phi) \vee (s \vdash \quad \Phi) \vee (s \vdash \quad \Phi)] = 1 \tag{5}$$

A model-checking algorithm that satisfies (2) through (5) is guaranteed to produce a correct answer with probability at least $1 - \alpha - \gamma$ when Φ holds and $1-\beta-\gamma$ when Φ does not hold. To make these probabilities high, α, β, and γ need to be low. If all three parameters are zero, then $\mathcal{A}$ is a deterministic algorithm for probabilistic model checking. If both $\alpha + \gamma$ and $\beta + \gamma$ are less than 0.5, but non-zero, then $\mathcal{A}$ is a randomized algorithm for probabilistic model checking.

Unfortunately, it is generally not possible, in practice, to satisfy all four conditions with low values for all three parameters. Next, we will discuss how these conditions are relaxed by current solution methods, and then we will present a new statistical solution method based on an alternative relaxation.

4 Current Solution Methods

Current solution methods, both numerical and statistical, can be seen as relying on a relaxation of (2) and (3) to become tractable. The reference point for error is changed from UTSL to UTSL$_\delta$ semantics, replacing (2) and (3) with:

$$\Pr[s \vdash \quad \Phi \mid s \approx^{\delta} \quad \Phi] \leq \alpha \tag{6}$$

$$\Pr[s \vdash \quad \Phi \mid s \approx^{\delta} \quad \Phi] \leq \beta \tag{7}$$

4.1 Statistical Hypothesis Testing

The predominant statistical solution method for verifying $\mathcal{P}_{\bowtie\theta}[\varphi]$ in a single state s is based on statistical *hypothesis testing*. This method was first proposed by Younes and Simmons [19] and further refined by Younes [17]. The approach always produces a definite result ($\gamma = 0$). This ensures a high probability of a correct answer when $s \mid\!\approx^{\delta} \Phi$ or $s \mid\!\approx^{\delta} \Phi$ holds.

Let Φ be $\mathcal{P}_{\geq\theta}[\varphi]$, let p be the probability measure of the set of trajectories that start in s and satisfy φ, and let X_i be Bernoulli variates with $\Pr[X_i = 1] = p$. To verify Φ we test the hypothesis $H_0 : p \geq \theta + \delta$ against the alternative hypothesis $H_1 : p \leq \theta - \delta$ based on observations of X_i (the result of verifying φ over a sample trajectory starting in s). Note that H_0 corresponds to $s \mid\!\approx^{\delta} \Phi$ and H_1 corresponds to $s \mid\!\approx^{\delta} \Phi$. If we take acceptance of H_0 to mean acceptance of Φ as true and acceptance of H_1 to mean rejection of Φ as false, then we can use *acceptance sampling* to verify Φ. Acceptance sampling is a well-established technique for statistical hypothesis testing. An acceptance sampling test with *strength* $\langle\alpha, \beta\rangle$ guarantees that H_1 is accepted with probability at most α when H_0 holds and H_0 is accepted with probability at most β when H_1 holds. Hence, we can use such a test to satisfy (6) and (7) for the verification of Φ.

Any acceptance sampling test with the prescribed strength can be used. A straightforward approach is to use a fixed number of observations $x_1, \ldots, x_n$ of the Bernoulli variates $X_1, \ldots, X_n$ and pick a constant c. If $\sum_{i=1}^{n} x_i$ is greater than c, then H_0 is accepted, otherwise H_1 is accepted. The pair $\langle n, c\rangle$ is called a *single sampling plan* [5]. The sum of n Bernoulli variates with parameter p has a binomial distribution with cumulative distribution function

$$F(c; n, p) = \sum_{i=0}^{c} \binom{n}{i} p^i (1-p)^{n-i} \ . \tag{8}$$

Using a single sampling plan $\langle n, c\rangle$ we accept hypothesis H_1 with probability $F(c; n, p)$ and hypothesis H_0 with probability $1 - F(c; n, p)$. To achieve strength $\langle\alpha, \beta\rangle$ we need to choose n and c so that $F(c; n, \theta+\delta) \leq \alpha$ and $1 - F(c; n, \theta-\delta) \leq \beta$. For optimal performance we choose n and c so that n is minimized. There is no closed-form solution for n, in general. Younes [17] describes an algorithm based on binary search that finds an optimal single sampling plan.

The sample size for a single sampling plan is fixed and therefore independent of the actual observations made. It is often possible to reduce the *expected* sample size required to achieve a desired test strength by taking the observations into account as they are made. This is called *sequential* acceptance sampling. Wald's [16] *sequential probability ratio test* (SPRT) is a particularly efficient sequential test. The reduction in expected sample size, compared to a single sampling plan, is often substantial, although there is no fixed upper bound on the sample size. The SPRT is carried out as follows. At the mth stage, i.e. after making m observations $x_1, \ldots, x_m$ we calculate the quantity

$$f_m = \prod_{i=1}^{m} \frac{\Pr[X_i = x_i \mid p = p_1]}{\Pr[X_i = x_i \mid p = p_0]} = \frac{p_1^{d_m}(1-p_1)^{m-d_m}}{p_0^{d_m}(1-p_0)^{m-d_m}} \ , \tag{9}$$

where $d_m = \sum_{i=1}^{m} x_i$. Hypothesis H_0 is accepted if $f_m \leq \beta/(1-\alpha)$, and hypothesis H_1 is accepted if $f_m \geq (1-\beta)/\alpha$. Otherwise, additional observations are made until either termination condition is satisfied.

4.2 Statistical Estimation

An alternative statistical solution method, based on *estimation* instead of hypothesis testing, has been developed by Lassaigne and Peyronnet [13]. Hérault et al. [8] provide more details of this approach.

As before, let Φ be $\mathcal{P}_{\geq\theta}[\varphi]$ and p the probability measure of the set of trajectories that start in s and satisfy φ. This approach uses n observations $x_1, \ldots, x_n$ to compute an estimate of p: $\tilde{p} = \frac{1}{n}\sum_{i=1}^{n} x_i$. The estimate is such that

$$\Pr\big[|\tilde{p} - p| < \delta\big] \geq 1 - \alpha \ . \tag{10}$$

Using a result derived by Hoeffding [10–Theorem 1], it can be shown that

$$n = \left\lceil \frac{1}{2\delta^2} \log \frac{2}{\alpha} \right\rceil \tag{11}$$

is sufficient to satisfy (10). If we accept Φ as true when $\tilde{p} \geq \theta$ and reject Φ as false otherwise, then it follows from (10) that the answer is correct with probability at least $1-\alpha$ if either $s \models_{\approx}^{\delta} \Phi$ or $s \models_{\approx}^{\delta} \Phi$ holds. Consequently, the verification procedure satisfies (6) and (7) with $\beta = \alpha$. As with the solution method based on hypothesis testing, a definite answer is always generated ($\gamma = 0$).

To compare the estimation-based approach with the approach based on hypothesis testing, let $c = \lfloor n\theta + 1 \rfloor$ and $d = n\tilde{p} = \sum_{i=1}^{n} x_i$. It should be clear that $\tilde{p} \geq \theta \iff d > c$. This means that the estimation-based approach can be interpreted as a single sampling plan $\langle n, c \rangle$. It follows that the approach proposed by Younes and Simmons [19], when using a single sampling plan, *will always be at least as efficient as the estimation-based approach.* Typically, it will be more efficient because: (i) the sample size is derived using the true underlying distribution, (ii) c is not restricted to be $\lfloor n\theta + 1 \rfloor$, and (iii) $\beta \neq \alpha$ can be accommodated. The last property, in particular, is important when dealing with conjunctive and nested probabilistic statements. The advantage of hypothesis testing is demonstrated in Table 1. Note, also, that the SPRT often can be used to improve efficiency even further for the approach based on hypothesis testing.

4.3 Numerical Transient Analysis

To verify the formula $\mathcal{P}_{\bowtie\theta}[\varphi]$ in some state s we can compute p—the probability measure of the set of trajectories that start in s and satisfy φ—numerically and test if $p \bowtie \theta$ holds.

For time-bounded properties ($\varphi = \Phi\ \mathcal{U}^{[0,\tau]}\ \Psi$), which are the focus of this paper, such numerical computation is primarily feasible for Markov chains. Let $\mathcal{M}$ be a continuous-time Markov chain. First, as initially proposed by Baier et al. [2], the problem is reduced to *transient analysis* of a modified Markov

Table 1. Sample sizes for estimation and optimal single sampling plan ($\delta = 10^{-2}$)

θ	α	β	n_{est}	n_{opt}	$n_{\text{est}}/n_{\text{opt}}$
0.5	10^{-2}	10^{-2}	26,492	13,527	1.96
0.5	10^{-8}	10^{-2}	95,570	39,379	2.43
0.5	10^{-8}	10^{-8}	95,570	78,725	1.21
0.9	10^{-2}	10^{-2}	26,492	4,861	5.45
0.9	10^{-8}	10^{-2}	95,570	13,982	6.84
0.9	10^{-8}	10^{-8}	95,570	28,280	3.38

chain $\mathcal{M}'$, where all states in $\mathcal{M}$ satisfying $\neg\Phi \vee \Psi$ have been made absorbing. Now, p is equal to the probability of occupying a state satisfying Ψ at time τ in model $\mathcal{M}'$. This probability can be computed using a technique called *uniformization*, originally proposed by Jensen [11]. Let $\mathbf{Q}$ be the generator matrix of $\mathcal{M}'$, $q = \max_i -q_{ii}$, and $\mathbf{P} = \mathbf{I} + \mathbf{Q}/q$. Then p can be expressed as follows [3]:

$$p = \vec{\mu}_0 \cdot \sum_{k=0}^{\infty} e^{-q\cdot\tau} \frac{(q\cdot\tau)^k}{k!} \mathbf{P}^k \cdot \vec{\chi}_\Psi \tag{12}$$

Here, $\vec{\mu}_0$ is a 0-1 row vector with a 1 in the column for the initial state s and $\vec{\chi}_\Psi$ is a 0-1 column vector with a 1 in each row corresponding to a state that satisfies Ψ.

In practice, the infinite summation in (12) is truncated by using the techniques of Fox and Glynn [6], so that the truncation error is bounded by ϵ. If $\tilde{p}$ is the computed probability, then $\tilde{p} \leq p \leq \tilde{p} + \epsilon$. It follows that by accepting $\mathcal{P}_{\bowtie\theta}[\varphi]$ as true if $\tilde{p} + \epsilon/2 \bowtie \theta$ and rejecting the formula as false otherwise, the numerical solution method satisfies (6) and (7) with $\delta = \epsilon/2$ and $\alpha = \beta = 0$. As with the statistical solution methods, a definite answer is always given ($\gamma = 0$). This shows that numerical and statistical solution methods for probabilistic model checking can, indeed, be viewed as solving the same problem, i.e. UTSL$_\delta$ model checking rather than UTSL model checking. Statistical solution methods are truly randomized algorithms for UTSL$_\delta$ model checking.

When using uniformization to verify $\mathcal{P}_{\geq\theta}[\Phi \; \mathcal{U}^{[0,\tau]} \; \Psi]$, it is actually possible to know when we cannot make an informed decision. If we accept the formula as true when $\tilde{p} \geq \theta$, reject it as false when $\tilde{p} + \epsilon < \theta$, and report "undecided" otherwise, then (2) and (3) can be satisfied with $\alpha = \beta = 0$. This alternative implementation of the numerical solution method no longer satisfies (4). That condition is replaced by $\Pr[s \vdash \; \Phi \mid (s \models \Phi) \vee (s \not\approx^\delta \; \Phi)] = 0$, with $\delta = \epsilon$, for $\mathcal{P}_{\geq\theta}[\varphi]$ without nested probabilistic operators, and

$$\Pr[s \vdash \; \Phi \mid (s \not\approx^\delta \; \Phi) \vee (s \not\approx^\delta \; \Phi)] = 0 \tag{13}$$

for an arbitrary formula Φ. The use of undecided results with numerical methods for probabilistic model checking has been suggested by Hermanns et al. [9], although it is not clear if any tool implements this approach. The leading tool for probabilistic model checking, PRISM [12], does not produce undecided results.

5 Statistical Solution Method with Undecided Results

Existing statistical solution methods provide no meaningful error bounds if neither $s \models_{\approx}^{\delta} \Phi$ nor $s \models_{\approx}^{\delta} \Phi$ holds. This section presents a new statistical solution method that satisfies (2) and (3), so whenever a definite result is given the probability of error is bounded. We accomplish this by allowing an undecided result with some probability. The goal is to replace (4) with

$$\Pr[s \vdash \Phi \mid (s \models_{\approx}^{\delta} \Phi) \vee (s \models_{\approx}^{\delta} \Phi)] \leq \gamma \ . \tag{14}$$

5.1 Probabilistic Operator Without Nesting

Let Φ be $\mathcal{P}_{\geq \theta}[\varphi]$ without nested probabilistic operators ($\mathcal{P}_{\leq \theta}[\varphi]$ is analogous). To satisfy (2), (3), and (14) simultaneously using a sample of size n we pick two constants c_0 and c_1 such that $0 \leq c_1 < c_0 < n$ and the following conditions hold:

$$F(c_1; n, \theta) \leq \alpha \tag{15}$$

$$1 - F(c_1; n, \theta - \delta) \leq \gamma \tag{16}$$

$$1 - F(c_0; n, \theta) \leq \beta \tag{17}$$

$$F(c_0; n, \theta + \delta) \leq \gamma \tag{18}$$

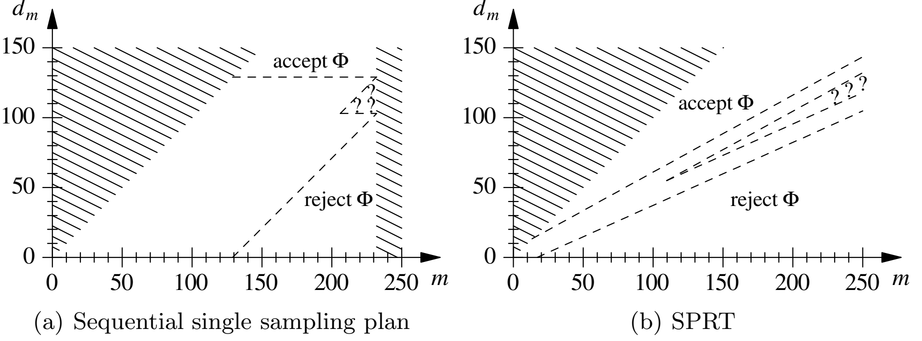
Let $d = \sum_{i=1}^{n} x_i$. Formula Φ is accepted as true if $d > c_0$ and rejected as false if $d \leq c_1$; otherwise ($c_1 < d \leq c_0$) the result is undecided.

The procedure just given can be interpreted as using *two simultaneous* acceptance sampling tests. The first is used to tests $H_0 : p \geq \theta$ against $H_1 : p \leq \theta - \delta$ with strength $\langle \alpha, \gamma \rangle$. The second is used to tests $H_0 : p \geq \theta + \delta$ against $H_1 : p \leq \theta$ with strength $\langle \gamma, \beta \rangle$. H_0 represents acceptance of Φ as true, H_1 represents rejection of Φ as false, and the remaining two hypotheses represent an undecided result. Combining the results from both tests, Φ is accepted as true if both H_0 and H_0 are accepted, Φ is rejected as false if both H_1 and H_1 are accepted, otherwise the result is undecided. Of course, this means that we do not need to use hypothesis testing with fixed-size samples. We could use any acceptance sampling plans with the prescribed strengths and combine their results as specified. In particular, we could use the SPRT to reduce the expected sample size.

Graphical representations of two acceptance sampling tests with undecided results are shown in Fig. 1 for $\theta = 0.5$, $\delta = 0.1$, $\alpha = 0.04$, $\beta = 0.08$, and $\gamma = 0.1$. The horizontal axis represents the number of observations and the vertical axis represents the number of positive observations. Figure 1(a) represents a sequential version of a single sampling plan with $n = 232$, $c_0 = 128$, $c_1 = 102$. The line $d_m = 129$ is the boundary for acceptance of Φ. There is a line for rejection of Φ and two lines defining the boundary of the region that represents an undecided result. Figure 1(b) shows the corresponding decision boundaries for the SPRT.

5.2 Composite Formulae

For a negation $\neg\Phi$ we have $s \vdash \neg\Phi \iff s \vdash \Phi$. Hence, if we can satisfy (14) for Φ, then we have the same bound, γ, on the probability of an undecided



(a) Sequential single sampling plan (b) SPRT

Fig. 1. Graphical representation of acceptance sampling tests

result for the negation of Φ. The roles of α and β are reversed for negation (cf. Younes and Simmons [19] and Younes [17]).

For a conjunction $\Phi \wedge \Psi$ we get the following general bound on the probability of an undecided result:

$$
\begin{aligned}
&\Pr[s \vdash \;\; \Phi \wedge \Psi \mid (s \not\approx^{\delta} \Phi \wedge \Psi) \vee (s \not\approx^{\delta} \Phi \wedge \Psi)] \\
&\quad \leq \max(\gamma_\Phi + \gamma_\Psi, \gamma_\Phi + \beta_\Phi, 2\gamma_\Psi + \beta_\Psi)
\end{aligned} \tag{19}
$$

In practice, the dependence on β_Φ and β_Ψ can be disregarded. We have β_Φ in (19) because $\Pr[s \vdash \;\; \Phi \mid s \not\approx^{\delta} \Phi] \leq \Pr[s \vdash \;\; \Phi \mid s \not\models \Phi] \leq \beta_\Phi$ (similarly for β_Ψ), but $\Pr[s \vdash \;\; \Phi \mid s \not\approx^{\delta} \Phi]$ is typically negligible compared to $\Pr[s \vdash \;\; \Phi \mid s \not\models \Phi]$. Let $\gamma' = \gamma_\Phi = \gamma_\Psi$. Then (19) can, *for all practical purposes*, be replaced by

$$
\Pr[s \vdash \;\; \Phi \wedge \Psi \mid (s \not\approx^{\delta} \Phi \wedge \Psi) \vee (s \not\approx^{\delta} \Phi \wedge \Psi)] \leq 2\gamma' \ . \tag{20}
$$

Consequently, if we want to ensure at most a γ probability of an undecided result for $\Phi \wedge \Psi$, and we use the same bound for both conjuncts, then we can use $\gamma/2$ when verifying Φ and Ψ. For a conjunction of size n, the symmetric bound for each conjunct could be set to γ/n.

To satisfy (2) we should choose α_Φ and α_Ψ such that $\alpha_\Phi + \alpha_\Psi \leq \alpha$ (cf. Younes and Simmons [19][1]):

$$
\begin{aligned}
&\Pr[(s \vdash \;\; \Phi) \vee (s \vdash \;\; \Psi) \mid (s \models \Phi) \wedge (s \models \Psi)] \\
&\quad \leq \Pr[s \vdash \;\; \Phi \mid s \models \Phi] + \Pr[s \vdash \;\; \Psi \mid s \models \Psi] \leq \alpha_\Phi + \alpha_\Psi
\end{aligned} \tag{21}
$$

Similar to γ, we can use α/n when verifying the parts of a conjunction of size n. Unlike γ, however, this does not involve any approximation. To satisfy (3), it suffices to use the same error bound, β, for the individual conjuncts:

$$
\begin{aligned}
&\Pr[(s \vdash \;\; \Phi) \wedge (s \vdash \;\; \Psi) \mid (s \not\models \Phi) \vee (s \not\models \Psi)] \\
&\quad \leq \max(\Pr[s \vdash \;\; \Phi \mid s \not\models \Phi], \Pr[s \vdash \;\; \Psi \mid s \not\models \Psi]) \leq \max(\beta_\Phi, \beta_\Psi)
\end{aligned} \tag{22}
$$

[1] Younes [17] gives the bound $\min(\alpha_\Phi, \alpha_\Psi)$, but this is a bound only *for each individual way* of rejecting a conjunction as false. The result due to Younes and Simmons [19] and reproduced here bounds the probability of rejecting a conjunction *in any way*.

5.3 Nested Probabilistic Statements

We use acceptance sampling to verify probabilistic statements. The observations that are used by the acceptance sampling test correspond to the verification of a path formula, φ, over sample trajectories. If φ contains probabilistic statements, then the observations may be incorrect or undecided. We assume that φ can be verified with parameters α_φ, β_φ, and γ_φ. This can be accomplished by treating the path formula as a large disjunction of conjunctions, as described by Younes and Simmons [19–p. 231] and Younes [17–p. 78].

It remains to show how to use the verification results for φ to verify a probabilistic statement, $\Phi = \mathcal{P}_{\geq \theta}[\varphi]$, so that (2), (3), and (14) are satisfied. This can be accomplished by a single sampling plan with n, c_0, and c_1 chosen to satisfy the following conditions:

$$F(c_1; n, \theta(1 - \alpha_\varphi)) \leq \alpha \tag{23}$$

$$1 - F(c_1; n, 1 - (1 - (\theta - \delta))(1 - \gamma_\varphi - \beta_\varphi)) \leq \gamma \tag{24}$$

$$1 - F(c_0; n, \theta + (1 - \theta)\beta_\varphi) \leq \beta \tag{25}$$

$$F(c_0; n, (\theta + \delta)(1 - \gamma_\varphi - \alpha_\varphi)) \leq \gamma \tag{26}$$

This assumes that Φ is accepted as true when more than c_0 positive observations are made, Φ is rejected as false when at most c_1 observations are non-positive (i.e., negative *or* undecided), and the result is undecided otherwise.

Compared to (15) through (18) for acceptance sampling without nested probabilistic operators, the only difference is that the probability thresholds have been modified. The indifference regions of the two acceptance sampling tests have been made narrower to account for the possibility of erroneous or undecided observations. We can use the same modification with the SPRT.

It should be noted that α_φ, β_φ and γ_φ can be chosen independently of α, β, and γ. The choice of parameters for the verification of φ is restricted only by the following conditions:

$$(\theta - \delta) + (1 - (\theta - \delta))(1 - \gamma_\varphi - \beta_\varphi) < \theta(1 - \alpha_\varphi) \tag{27}$$

$$\theta + (1 - \theta)\beta_\varphi < (\theta + \delta)(1 - \gamma_\varphi - \alpha_\varphi) \tag{28}$$

The choice of α_φ, β_φ, and γ_φ can have a significant impact on performance (cf. the discussion by Younes [17] regarding the impact of observation error on performance for the standard statistical solution method).

6 Complexity of Statistical Solution Methods

The time complexity of any statistical solution method for probabilistic model checking can be understood in terms of two main factors: the sample size and the length of sample trajectories. The sample size depends on the method used for verifying probabilistic statements and the desired strength. The length of trajectories depends on model characteristics and the property that is being

verified. An additional factor is simulation effort, which can be both model and implementation dependent.

Consider the formula $\mathcal{P}_{\bowtie\theta}[\Phi\,\mathcal{U}^{[0,\tau]}\,\Psi]$ without nested probabilistic operators. Let q be the expected number of state transitions per time unit, let m be the simulation effort per state transition, and let N be the sample size. The time complexity of statistical probabilistic model checking for the given formula is $O(q \cdot \tau \cdot m \cdot N)$. The sample size, N, is the only factor that varies between different statistical solution methods, regardless of implementation details.

If we use a single sampling plan with strength $\langle\alpha, \beta\rangle$ and indifference region of half-width δ, then N is roughly proportional to $\log\alpha$ and $\log\beta$ and inversely proportional to δ^2 [17–p. 23]. We have shown in this paper that the approach based on statistical estimation described by Hérault et al. [8] never uses a smaller sample size than a single sampling plan, given the same parameters, and often uses a much larger sample size. Using the SPRT instead of a single sampling plan can reduce the expected sample size by orders of magnitude in most cases, although the SPRT is not guaranteed always to be more efficient (this is well known in the statistics literature; Younes [17] provides examples of this in the context of model checking). The new statistical approach presented in this paper, which can produce undecided results, has the same time complexity as the old statistical solution method. Given the same α, β, and δ, the new method will require a larger sample size because it is based on acceptance sampling with indifference regions of half-width $\delta/2$, instead of δ for the old method.

Results presented by Sen et al. [15] give the impression that single sampling plans consistently outperform the SPRT. It should be noted, however, that Sen et al. manually selected the sample sizes for their single sampling plans, guided by a desire to achieve a low p-value (K. Sen, personal communication, May 20, 2004). The selected sample sizes are not sufficient to achieve the same strength as used to produce the results for the SPRT reported by Younes et al. [18], on which they base their comparison. All their empirical evaluation really proves is that a smaller sample size results in shorter verification time—which should surprise no one—but the casual reader may be misled into believing that Sen et al. have devised a more efficient statistical solution method.

7 Empirical Evaluation

The performance of our new statistical solution method is similar to that of the previous statistical solution method, which has been studied and compared to the numerical approach by Younes et al. [18]. We limit the empirical evaluation in this paper to a brief study of the effect that the parameter γ has on performance.

Figure 2 plots the *expected* sample size, as a function of the (unknown) probability p that a path formula holds, for the SPRT and a *sequential* single sampling plan (SSSP) with different parameter choices ($\theta = 0.5$, $\delta = 0.1$, $\alpha_\Delta = 0.004$, $\alpha_\nabla = 0.04$, $\beta_\Delta = 0.008$, $\beta_\nabla = 0.08$, $\gamma_\Delta = 0.01$, and $\gamma_\nabla = 0.1$). The expected sample size is low outside of the indifference region (gray area), especially for the SPRT, and peaks in the indifference region. Note the drop in expected sample

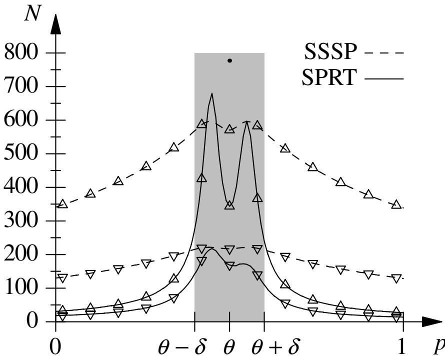


Fig. 2. Expected sample size

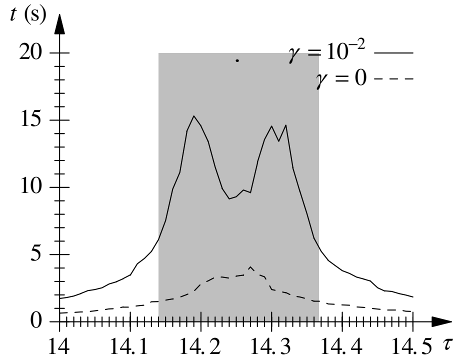


Fig. 3. Verification time

size at the threshold θ where an undecided result is given with high probability. The expected sample size, as a function of p, will be similar for other parameter values, with the SPRT almost always outperforming a (sequential) single sampling plan by a wide margin.

Now, consider the model-checking problem for an n-station symmetric polling system used by Younes et al. [18]. Each station has a single-message buffer and the stations are attended by a single server in cyclic order. The server begins by polling station 1. If there is a message in the buffer of station 1, the server starts serving that station. Once station i has been served, or if there is no message at station i when it is polled, the server starts polling station $i+1$ (or 1 if $i = n$). We verify the property $m_1{=}1 \rightarrow \mathcal{P}_{\geq 0.5}[\top \ \mathcal{U}^{[0,\tau]} \ poll_1]$, which states that if station 1 is full, then it is polled within τ time units with probability at least 0.5. We do so in the state where station 1 has just been polled and all buffers are full.

Figure 3 plots the verification time for the symmetric polling system problem ($n = 10$), as a function of the formula time bound τ, averaged over 100 runs. The plot shows the verification time for the new solution method with $\gamma = 10^{-2}$ (solid curve) and the old solution method without undecided results (dashed curve); $2\delta = 10^{-2}$ and $\alpha = \beta = 10^{-2}$ in both cases. The verification time is lower for the standard statistical solution method, but it produces more erroneous results. Table 2 shows the number of times a certain result is produced for seven different values of τ. The new statistical solution method does not produce an erroneous result in any of the experiments, while the error probability is high for

Table 2. Result distribution *with* (bottom) and *without* (top) undecided results

result	**14.10**	**14.15**	**14.20**	**14.25**	**14.30**	**14.35**	**14.40**
accept	0	3	9	50	88	97	100
reject	100	97	91	50	12	3	0
accept	0	0	0	0	32	99	100
reject	100	99	42	1	0	0	0
undecided	0	1	58	99	68	1	0

the standard statistical solution method for values of τ close to 14.251 (where the value of the verified property goes from false to true). Higher reliability in the results are obtained at the cost of efficiency.

8 Discussion

We have presented a framework for expressing correctness guarantees of model-checking algorithms. Using this framework, we have shown how current solution methods for probabilistic model checking are related. In particular, we have shown that Younes and Simmons' [19] statistical solution method based on hypothesis testing has clear benefits over Hérault et al.'s [8] estimation-based approach, and that numerical and statistical solution methods can be interpreted as solving the same *relaxed* model-checking problems. In addition, we have presented a new statistical solution method that bounds the probability of error under all circumstances. This is accomplished by permitting undecided results, and we have shown how to guarantee bounds for the probability of getting an undecided result for any time-bounded formula.

References

1. Alur, R. and Dill, D. L. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. Baier, C., Haverkort, B. R., Hermanns, H., and Katoen, J.-P. Model checking continuous-time Markov chains by transient analysis. In *Proc. 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 358–372. Springer, 2000.
3. Baier, C., Haverkort, B. R., Hermanns, H., and Katoen, J.-P. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.
4. Clarke, E. M. and Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. 1981 Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.
5. Duncan, A. J. *Quality Control and Industrial Statistics.* Richard D. Irwin, fourth edition, 1974.
6. Fox, B. L. and Glynn, P. W. Computing Poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988.
7. Hansson, H. and Jonsson, B. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
8. Hérault, T., Lassaigne, R., Magniette, F., and Peyronnet, S. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 73–84. Springer, 2004.
9. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., and Siegle, M. A tool for model-checking Markov chains. *International Journal on Software Tools for Technology Transfer*, 4(2):153–172, 2003.
10. Hoeffding, W. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

11. Jensen, A. Markoff chains as an aid in the study of Markoff processes. *Skandinavisk Aktuarietidskrift*, 36:87–91, 1953.
12. Kwiatkowska, M., Norman, G., and Parker, D. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.
13. Lassaigne, R. and Peyronnet, S. Approximate verification of probabilistic systems. In *Proc. 2nd Joint International PAPM-PROBMIV Workshop*, volume 2399 of *LNCS*, pages 213–214. Springer, 2002.
14. Segala, R. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995. MIT-LCS-TR-676.
15. Sen, K., Viswanathan, M., and Agha, G. Statistical model checking of black-box probabilistic systems. In *Proc. 16th International Conference on Computer Aided Verification*, volume 3114 of *LNCS*, pages 202–215. Springer, 2004.
16. Wald, A. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945.
17. Younes, H. L. S. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, 2005. CMU-CS-05-105.
18. Younes, H. L. S., Kwiatkowska, M., Norman, G., and Parker, D. Numerical vs. statistical probabilistic model checking: An empirical study. In *Proc. 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 46–60. Springer, 2004.
19. Younes, H. L. S. and Simmons, R. G. Probabilistic verification of discrete event systems using acceptance sampling. In *Proc. 14th International Conference on Computer Aided Verification*, volume 2404 of *LNCS*, pages 223–235. Springer, 2002.

Field Constraint Analysis

Thomas Wies[1], Viktor Kuncak[2], Patrick Lam[2],
Andreas Podelski[1], and Martin Rinard[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{wies, podelski}@mpi-inf.mpg.de
[2] MIT Computer Science and Artificial Intelligence Lab, Cambridge, USA
{vkuncak, plam, rinard}@csail.mit.edu

Abstract. We introduce *field constraint analysis*, a new technique for verifying data structure invariants. A field constraint for a field is a formula specifying a set of objects to which the field can point. Field constraints enable the application of decidable logics to data structures which were originally beyond the scope of these logics, by verifying the backbone of the data structure and then verifying constraints on fields that cross-cut the backbone in arbitrary ways. Previously, such cross-cutting fields could only be verified when they were uniquely determined by the backbone, which significantly limits the range of analyzable data structures.

Field constraint analysis permits *non-deterministic* field constraints on cross-cutting fields, which allows the verificiation of invariants for data structures such as skip lists. Non-deterministic field constraints also enable the verification of invariants between data structures, yielding an expressive generalization of static type declarations.

The generality of our field constraints requires new techniques. We present one such technique and prove its soundness. We have implemented this technique as part of a symbolic shape analysis deployed in the context of the Hob system for verifying data structure consistency. Using this implementation we were able to verify data structures that were previously beyond the reach of similar techniques.

1 Introduction

The goal of shape analysis [27, Chapter 4], [6, 32, 26, 2, 4, 25, 5, 22] is to verify complex consistency properties of linked data structures. The verification of such properties is important in itself, because the correct execution of the program often requires data structure consistency. In addition, the information computed by shape analysis is important for verifying other program properties in programs with dynamic memory allocation.

Shape analyses based on expressive decidable logics [26, 14, 12] are interesting for several reasons. First, the correctness of such analyses is easier to establish than for approaches based on ad-hoc representations; the use of a decidable logic separates the problem of generating constraints that imply program properties from the problem of solving these constraints. Next, such analyses can be used in the context of assume-guarantee reasoning because logics provide a language for specifying the behaviors of code fragments. Finally, the decidability of logics leads to completeness properties for these analyses, eliminating false alarms and making the analyses easier to interact with. We were able to confirm these observations in the context of Hob system [21, 16] for analyzing data structure consistency, where we have integrated one such shape analysis

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 157–173, 2006.
© Springer-Verlag Berlin Heidelberg 2006

[26] with other analyses, allowing us to use shape analysis in the context of larger programs: in particular, Hob enabled us to leverage the power of shape analysis, while avoiding the associated performance penalty, by applying shape analysis only to those parts of the program where its extreme precision is necessary.

Our experience with such analyses has also taught us that some of the techniques that make these analyses predictable also make them inapplicable to many useful data structures. Among the most striking examples is the restriction on pointer fields in the Pointer Assertion Logic Engine [26]. This restriction states that all fields of the data structure that are not part of the data structure's tree backbone must be functionally determined by the backbone; that is, such fields must be specified by a formula that uniquely determines where they point to. Formally, we have

$$\forall x\ y.\ f(x){=}y \ \leftrightarrow\ F(x,y) \tag{1}$$

where f is a function representing the field, and F is the defining formula for f. The relationship (1) means that, although data structures such as doubly linked lists with backward pointers can be verified, many other data structures remain beyond the scope of the analysis. This includes data structures where the exact value of pointer fields depends on the history of data structure operations, and data structures that use randomness to achieve good average-case performance, such as skip lists [30]. In such cases, the invariant on the pointer field does not uniquely determine where the field points to, but merely gives a constraint on the field, of the form

$$\forall x\ y.\ f(x){=}y \ \rightarrow\ F(x,y) \tag{2}$$

This constraint is equivalent to $\forall x.\ F(x, f(x))$, which states that the function f is a solution of a given binary predicate. The motivation for this paper is to find a technique that supports reasoning about constraints of this, more general, form. In a search for existing approaches, we have considered structure simulation [11,9], which, intuitively, allows richer logics to be embedded into existing logics that are known to be decidable, and of which [26] can be viewed as a specific instance. Unfortunately, even the general structure simulation requires definitions of the form $\forall x\ y.\ r(x,y) \ \leftrightarrow\ F(x,y)$ where $r(x,y)$ is the relation being simulated. To handle the general case (2), an alternative approach therefore appears to be necessary.

Field constraint analysis. This paper presents field constraint analysis, our approach for analyzing fields with general constraints of the form (2). Field constraint analysis is a proper generalization of the existing approach and reduces to it when the constraint formula F is functional. It is based on approximating the occurrences of f with F, taking into account the polarity of f, and is always sound. It is expressive enough to verify constraints on pointers in data structures such as two-level skip lists. The applicability of our field constraint analysis to non-deterministic field constraints is important because many complex properties have useful non-deterministic approximations. Yet despite this fundamentally approximate nature of field constraints, we were able to prove its completeness for some important special cases. Field constraint analysis naturally combines with structure simulation, as well as with a symbolic approach to shape analysis [33,29]. Our presentation and current implementation are in the context of the monadic second-order logic (MSOL) of trees [13], but our results extend to other log-

ics. We therefore view field constraint analysis as a useful component of shape analysis approaches that makes shape analysis applicable to a wider range of data structures.

Contributions. This paper makes the following contributions:

- We introduce an **algorithm** (Figure 9) that uses field constraints to eliminate derived fields from verification conditions.
- We prove that the algorithm is both **sound** (Theorem 1) and, in certain cases, **complete**. The completeness applies not only to deterministic fields (Theorem 2), but also to the preservation of field constraints themselves over loop-free code (Theorem 3). Theorem 3 implies a complete technique for checking that field constraints hold, if the programmer adheres to a discipline of maintaining them, for instance at the beginning of each loop.
- We describe how to combine our algorithm with symbolic shape analysis [33] to **infer loop invariants**.
- We describe an **implementation** and experience in the context of the Hob system for verifying data structure consistency. The implementation of field constraint analysis as part of the Hob system [21, 16] allows us to apply the analysis to modules of larger applications, with other modules analyzed by more scalable analyses, such as typestate analysis [20].

Additional details (including proofs of theorems) are in [34].

2 Examples

We next explain our field constraint analysis with a set of examples. Note that our analysis handles, as a special case, data structures that have back pointers constrained by deterministic constraints. Such data structures (for instance, doubly linked lists and trees with parent pointers [34]) have also been analyzed by previous approaches [26]. To illustrate the additional power of our analysis, we first present an example illustrating inter-data-structure constraints, which are simple and useful for high-level application properties, but are often nondeterministic. We then present a skip list example, which shows how non-deterministic field constraints arise within data structures, and illustrates how our analysis can synthesize loop invariants.

2.1 Students and Schools

The data structure in our first example contains two linked lists: one containing students and one containing schools (Figure 2). Each `Elem` object may represent either a student or a school; students have a pointer to the school which they attend. Both students and schools use the `next` backbone pointer to indicate the next student or school in the relevant linked list. An invariant of the data structure is that, if an object is in the list of students, then its `attends` field points to an object in the schools list; that is, it cannot be null and it cannot point to an object outside the list of schools. This invariant is an example of a non-deterministic field constraint: the `attends` field has a non-trivial constraint, but the target of the field is not uniquely defined in terms of existing fields; instead, this field carries important new information about the school that each student attends.

We implement our example as a module in the Hob system [21], which allows us to specify and, using field constraint analysis, verify the desired data structure invariants and interfaces of data structure operations. In general, a module in Hob consists of three sections: 1) an implementation section (Figure 1) containing declarations of memory cell formats (in this case `Elem`) and executable code for data structure operations (such as `addStudent`); 2) a specification section (Figure 3) containing declarations of abstract sets of objects (such as `ST` for the set of students and `SC` for the set of schools in the data structure) and interfaces of data structure operations expressed in terms of these abstract sets; and 3) the abstraction section, which gives the abstraction function specifying the definition of sets (`SC` and `ST`) and specifies the representation invariants of the data structure, including field constraints (in this case, on the field `attends`).

The implementation in Figure 1 states that the `addStudent` procedure adds a student `st` to the student list and associates it (via the `attends` field) with an existing school `sc`, which is expected to be already in the list of schools. Figure 3 presents the set interface for the `addStudents` procedure, consisting of a precondition (`requires` clause), frame condition (`modifies` clause), and postcondition (`ensures` clause). The precondition states that `st` must not already be in the list of students `ST`, and that `sc` must be in the list of schools. We represent parameters as sets of cardinality at most one (the null object is represented as an empty set). Therefore, the conjuncts `card(st)=1` and `card(sc)=1` in the precondition indicate that the parameters `st` and `sc` are not null. The modifies clause indicates that only the set of students `ST` and not the set of schools `SC` is modified. The postcondition describes the effect of the procedure: it states that the set of students `ST'` after procedure execution is equal to the union (denoted +) of the set `ST` of student objects before procedure execution, and (the singleton containing) the given student object `st`.

Our analysis automatically verifies that the data structure operation `addStudent` conforms to its interface expressed in terms of abstract sets. Proving the conformance of a procedure to such a set interface is useful for several reasons. First, the preconditions indicate to data structure clients the conditions under which it is possible to invoke operations. These preconditions are necessary to prove that the field constraint is maintained: if it was not the case that the school parameter `sc` belonged to the set `SC` of schools, the insertion would violate the representation invariant. Similarly, if it was the case that the student object `st` was a member of the student list, insertion would introduce cycles in the list and violate the implicit acyclicity invariant of the data structure. Also, the postcondition of `addStudents` communicates the fact that `st` is in the list after the insertion, preventing clients from executing duplicate calls to `addStudents` with the same student object. Finally, the set interface expresses an important partial correctness property for the `addStudent` procedure, so that the verification of the set interface indicates that the procedure is correctly inserting an object into the set of students.

Note that the interface of the procedure does not reveal the details of procedure implementation, thanks to the use of abstract set variables. Since the set variables in the specification are abstract, any verification of a concrete implementation's conformance to the set interface requires concrete definitions for the abstract variables. The abstraction section in Figure 4 contains this information. First, the abstraction section

```
impl module Students {
  format Elem {
    attends : Elem;
    next : Elem;
  }
  var students : Elem;
  var schools  : Elem;

  proc addStudent(st:Elem; sc:Elem) {
    st.attends = sc;
    st.next = students;
    students = st;
  }
}
```

Fig. 1. Implementation for students example

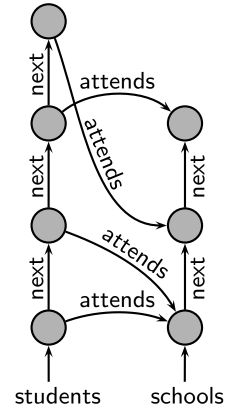


Fig. 2. Students data structure instance

```
spec module Students {
  format Elem;
  specvar ST : Elem set;
  specvar SC : Elem set;

  proc addStudent(st:Elem; sc:Elem)
    requires card(st)=1 & card(sc)=1 & (sc in SC) &
             (not (st in ST)) & (not (st in SC))
    modifies ST
    ensures ST' = ST + st;
}
```

Fig. 3. Specification for students example

```
abst module Students {
  use plugin "Bohne decaf";

  ST = { x : Elem | "rtrancl (% v1 v2. next v1 = v2) students x" };
  SC = { x : Elem | "rtrancl (% v1 v2. next v1 = v2) schools x" };

  invariant "ALL x y. (attends x = y) -->
      (x ~= null -->
      ((~(rtrancl (% v1 v2. next v1 = v2) students x) --> y = null) &
      ((rtrancl (% v1 v2. next v1 = v2) students x) -->
      (rtrancl (% v1 v2. next v1 = v2) schools y))))";

  invariant "ALL x.
      (x ~= null & (rtrancl (% v1 v2. next v1 = v2) schools x) -->
      ~(rtrancl (% v1 v2. next v1 = v2) students x))";

  invariant "ALL x.
      (x ~= null & (rtrancl (% v1 v2. next v1 = v2) students x) -->
      ~(rtrancl (% v1 v2. next v1 = v2) schools x))";
    ...
}
```

Fig. 4. Abstraction for students example

indicates which analysis (in this case, `Bohne decaf`, which implements field constraint analysis) is to be used to analyze the module. Next, the abstraction section contains definitions for abstract variables: namely, `ST` is defined as the set of `Elem` objects reachable from the root `students` through `next` fields, and `SC` is the set of `Elem` objects reachable from `schools`. (The function `rtrancl` is a higher-order function that accepts a binary predicate on objects and returns the reflexive transitive closure of the predicate.) The abstraction section also specifies data structure invariants, including field constraints. Field constraints are invariants with syntactic form `ALL x y. (f x = y) --> ···`. A field `f` for which there is no field constraint invariant in the abstraction section is considered to be part of the data structure *backbone*, which has an implicit invariant that it is a union of trees. Finally, the abstraction section may contain additional invariants; our example contains invariants stating disjointness of the lists rooted at `students` and `schools`.

Our Bohne analysis verifies the conformance of a procedure to its specification as follows. It first desugars the modifies clauses into a frame formula and conjoins it with the ensures clause, then replaces abstract sets in preconditions and postconditions with their definitions from the abstraction section, obtaining a procedure contract in terms of the concrete state variables (`next` and `attends`). It then conjoins representation invariants of the data structure to preconditions and postconditions. For a loop-free procedure such as `addStudents`, the analysis can then generate a verification condition whose validity implies that the procedure conforms to its interface.

The generated verification condition for our example cannot directly be solved using decision procedures such as MONA: it contains the function symbol `attends` that violates the tree invariant required by MONA. Section 3 describes how our analysis uses field constraints in the verification condition to verify the validity of such verification conditions. Our analysis can successfully verify the property that for any student, `attends` points to some (undetermined) element of the `SC` set of schools. Note that this goes beyond the power of previous analyses, which required that the identity of the school pointed to by the student be functionally determined by the identity of the student. The example therefore illustrates how our analysis eliminates a key restriction of previous approaches—certain data structures exhibit properties that the logics in previous approaches were not expressive enough to capture.

2.2 Skip List

We next present the analysis of a two-level skip list. Skip lists [30] support logarithmic average-time access to elements by augmenting a linked list with sublists that skip over some of the elements in the list. The two-level skip list is a simplified implementation of a skip list with only two levels: the list containing all elements, and a sublist of this list. Figure 5 presents an example two-level skip list. Our implementation uses the `next` field to represent the main list, which forms the backbone of the data structure, and uses the derived `nextSub` field to represent a sublist of the main list. We focus on the `add` procedure, which inserts an element into an appropriate position in the skip list. Figure 6 presents the implementation of `add`, which first searches through `nextSub` links to get an estimate of the position of the entry, then finds the entry by searching through `next` links, and inserts the element into the main `next`-linked list. Optionally, the procedure

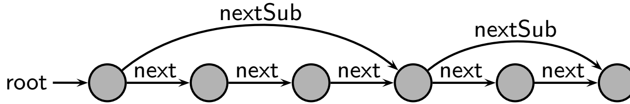


Fig. 5. An instance of a two-level skip list

```
impl module Skiplist {
  format Entry {
    v : int;
    next, nextSub : Entry;
  }
  var root : Entry;

  proc add(e:Entry) {
    assume "e ~= root";
    int v = e.v;
    Entry sprev = root, scurrent = root.nextSub;
    while ((scurrent != null) && (scurrent.v < v)) {
      sprev = scurrent; scurrent = scurrent.nextSub;
    }
    Entry prev = sprev, current = sprev.next;
    while ((current != scurrent) && (current.v < v)) {
      prev = current; current = current.next;
    }
    e.next = current; prev.next = e;
    choice { sprev.nextSub = e; e.nextSub = scurrent; }
         | { e.nextSub = null; }
}
```

Fig. 6. Skip list implementation

```
spec module Skiplist {
  format Entry;
  specvar S : Entry set;

  proc add(e:Entry)
    requires card(e) = 1 & not (e in S)
    modifies S
    ensures S' = S + e';
}
```

Fig. 7. Skip list specification

```
abst module Skiplist {
  use plugin "Bohne";

  S = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) (next root) x"};
  invariant "ALL x y. (nextSub x = y) --> ((x = null --> y = null) &
                (x ~= null --> rtrancl (% v1 v2. next v1 = v2) (next x) y))";
  invariant "root ~= null";
  invariant "ALL x. x ~= null &
              ~(rtrancl (% v1 v2. next v1 = v2) root x) -->
              ~(EX y. y ~= null & next y = x) & (next x = null)";
  proc add {
    has_pred = {x : Entry | "EX y. next y = x"};
    r_current = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) current x"};
    r_scurrent = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) scurrent x"};
    r_sprev = {x : Entry | "rtrancl (% v1 v2. next v1 = v2) sprev x"};
    next_null = {x : Entry | "next x = null"};
    sprev_nextSub = {x : Entry | "nextSub sprev = scurrent"};
    prev_next = {x : Entry | "next prev = current"};
  }
}
```

Fig. 8. Skip list abstraction (including invariants)

also inserts the element into `nextSub` list, which is modelled using a non-deterministic choice in our language and is an abstraction of the insertion with certain probability in the original implementation. Figure 7 presents a specification for `add`, which indicates that `add` always inserts the element into the set of elements stored in the list. Figure 8 presents the abstraction section for the two-level skip list. This section defines the abstract set S as the set of nodes reachable from `root.next`, indicating that `root` is used as a header node. The abstraction section contains three invariants. The first invariant is the field constraint on the field `nextSub`, which defines it as a derived field.

Note that the constraint for this derived field is non-deterministic, because it only states that if `x.nextSub==y`, then there exists a path of length at least one from `x` to `y` along `next` fields, without indicating where `nextSub` points. Indeed, the simplicity of the skip list implementation stems from the fact that the position of `nextSub` is not uniquely given by `next`; it depends not only on the history of invocations, but also on the random number generator used to decide when to introduce new `nextSub` links. The ability to support such non-deterministic constraints is what distinguishes our approach from approaches that can only handle deterministic fields.

The last two invariants indicate that `root` is never null (assuming, for simplicity of the example, that the state is initialized), and that all objects not reachable from `root` are isolated: they have no incoming or outgoing `next` pointers. These two invariants allow the analysis to conclude that the object referenced by `e` in `add(e)` is not referenced by any node, which, together with the precondition `not(e in S)`, allows our analysis to prove that objects remain in an acyclic list along the `next` field.[1]

Our analysis successfully verifies that `add` preserves all invariants, including the non-deterministic field constraint on `nextSub`. While doing so, the analysis takes advantage of these invariants as well, as is usual in assume/guarantee reasoning. In this example, the analysis is able to infer the loop invariants in `add`. The analysis constructs these loop invariants as disjunctions of universally quantified boolean combinations of unary predicates over heap objects, using symbolic shape analysis [33,29]. These unary predicates correspond to the sets that are supplied in the abstraction section using the `proc` keyword.

3 Field Constraint Analysis

This section presents the field constraint analysis algorithm and proves its soundness as well as, for some important cases, completeness.

We consider a logic $\mathcal{L}$ over a signature Σ where Σ consists of unary function symbols $f \in \mathsf{Fld}$ corresponding to fields in data structures and constant symbols $c \in \mathsf{Var}$ corresponding to program variables. We use monadic second-order logic (MSOL) of trees as our working example, but in general we only require $\mathcal{L}$ to support conjunction, implication, and equality reasoning.

[1] The analysis still needs to know that `e` is not identical to the header node. In this example we have used an explicit `(assume "e ≠ root")` statement to supply this information. Such assume statements can be automatically generated if the developer specifies the set of representation objects of a data structure, but this is orthogonal to field constraint analysis itself.

A Σ-structure S is a first-order interpretation of symbols in Σ. For a formula F in $\mathcal{L}$, we denote by $\mathsf{Fields}(F) \subseteq \Sigma$ the set of all fields occurring in F.

We assume that $\mathcal{L}$ is decidable over some set of well-formed structures and we assume that this set of structures is expressible by a formula I in $\mathcal{L}$. We call I the *simulation invariant* [11]. For simplicity, we consider the simulation itself to be given by the restriction of a structure to the fields in $\mathsf{Fields}(I)$, i.e. we assume that there exists a decision procedure for checking validity of implications of the form $I \rightarrow F$ where F is a formula such that $\mathsf{Fields}(F) \subseteq \mathsf{Fields}(I)$. In our running example, MSOL of trees, the simulation invariant I states that the fields in $\mathsf{Fields}(I)$ span a forest.

We call a field $f \in \mathsf{Fields}(I)$ a *backbone field*, and call a field $f \in \mathsf{Fld} \setminus \mathsf{Fields}(I)$ a *derived field*. We refer to the decision procedure for formulas with fields in $\mathsf{Fields}(I)$ over the set of structures defined by the simulation invariant I as *the underlying decision procedure*. Field constraint analysis enables the use of the underlying decision procedure to reason about non-deterministically constrained derived fields. We state invariants on the derived fields using field constraints.

Definition 1 (Field constraints on derived fields). *A* field constraint D_f *for a simulation invariant I and a derived field f is a formula of the form*

$$\mathsf{D}_f \;\equiv\; \forall x\, y.\; f(x) = y \;\rightarrow\; \mathsf{FC}_f(x, y)$$

where FC_f *is a formula with two free variables such that (1)* $\mathsf{Fields}(\mathsf{FC}_f) \subseteq \mathsf{Fields}(I)$, *and (2)* FC_f *is total with respect to I, i.e.* $I \models \forall x.\, \exists y\, .\, \mathsf{FC}_f(x, y)$. *We call the constraint* D_f deterministic *if* FC_f *is deterministic with respect to I, i.e.*

$$I \models \forall x\, y\, z.\; \mathsf{FC}_f(x, y) \wedge \mathsf{FC}_f(x, z) \;\rightarrow\; y = z\;\; .$$

We write D for the conjunction of D_f *for all derived fields f.*

Note that Definition 1 covers arbitrary constraints on a field, because D_f is equivalent to $\forall x.\; \mathsf{FC}_f(x, f(x))$.

The totality condition (2) is not required for the soundness of our approach, only for its completeness, and rules out invariants equivalent to “false”. The condition (2) does not involve derived fields and can therefore be checked automatically using a single call to the underlying decision procedure.

Our goal is to check validity of formulas of the form $I \wedge D \rightarrow G$, where G is a formula with possible occurrences of derived fields. If G does not contain any derived fields then there is nothing to do, because we can answer the query using the underlying decision procedure. To check validity of $I \wedge D \rightarrow G$, we therefore proceed as follows. We first obtain a formula G' from G by eliminating all occurrences of derived fields in G. Next, we check validity of G' with respect to I. In the case of a derived field f that is defined by a deterministic field constraint, occurrences of f in G can be eliminated by flattening the formula and substituting each term $f(x) = y$ by $\mathsf{FC}_f(x, y)$. However, in the general case of non-deterministic field constraints such a substitution is only sound for negative occurrences of derived fields, since the field constraint gives an over-approximation of the derived field. Therefore, a more sophisticated elimination algorithm is needed.

Eliminating derived fields. Figure 9 presents our algorithm Elim for elimination of derived fields. Consider a derived field f. The basic idea of Elim is that we can replace an occurrence $G(f(x))$ of f by a new variable y that satisfies $\mathsf{FC}_f(x, y)$, yielding a stronger formula $\forall y. \mathsf{FC}_f(x, y) \rightarrow G(y)$. As an improvement, if G contains two occurrences $f(x_1)$ and $f(x_2)$, and if x_1 and x_2 evaluate to the same value, then we attempt to replace $f(x_1)$ and $f(x_2)$ with the same value. Elim implements this idea using the set K of triples (x, f, y) to record previously assigned values for $f(x)$. Elim runs in time $O(n^2)$, where n is the size of the formula, and produces an at most quadratically larger formula. Elim accepts formulas in negation normal form, where all negation signs apply to atomic formulas. We generally assume that each quantifier $Q\, z$ binds a variable z that is distinct from other bound variables and distinct from the free variables of the entire formula. The algorithm Elim is presented as acting on first-order formulas, but is also applicable to checking validity of quantifier-free formulas because it only introduces universal quantifiers which can be replaced by Skolem constants. The algorithm is also applicable to multisorted logics, and, by treating sets of elements as a new sort, to MSOL. To make the discussion simpler, we consider a deterministic version of Elim where the non-deterministic choices of variables and terms are resolved by some arbitrary, but fixed, linear ordering on terms. We write $\mathsf{Elim}(G)$ to denote the result of applying Elim to a formula G.

S – a term or a formula
$\mathsf{Terms}(S)$ – terms occurring in S
$\mathsf{FV}(S)$ – variables free in S
$\mathsf{Ground}(S) = \{t \in \mathsf{Terms}(S).\ \mathsf{FV}(t) \subseteq \mathsf{FV}(S)\}$
$\mathsf{Derived}(S)$ – derived function symbols in S

proc $\mathsf{Elim}(G) = \mathsf{elim}(G, \emptyset)$
proc $\mathsf{elim}(G$: formula in negation normal form;
  K : set of (variable,field,variable) triples)**:**
  let $T = \{f(t) \in \mathsf{Ground}(G).\ f \in \mathsf{Derived}(G) \wedge \mathsf{Derived}(t) = \emptyset\}$
  if $T \neq \emptyset$ **do**
    choose $f(t) \in T$
    choose x, y fresh first-order variables
    let $F_1 = \mathsf{FC}_f(x, y) \wedge \bigwedge_{(x_i, f, y_i) \in K} (x = x_i \rightarrow y = y_i)$
    let $G_1 = G[f(t) := y]$
    return $\forall x.\ x = t \rightarrow \forall y.\ (F_1 \rightarrow \mathsf{elim}(G_1, K \cup \{(x, f, y)\}))$
  else case G **of**
  | $Qx.\ G_1$ **where** $Q \in \{\forall, \exists\}$**:**
    return $Qx.\ \mathsf{elim}(G_1, K)$
  | G_1 *op* G_2 **where** *op* $\in \{\wedge, \vee\}$**:**
    return $\mathsf{elim}(G_1, K)$ *op* $\mathsf{elim}(G_2, K)$
  | **else return** G

Fig. 9. Derived-field elimination algorithm

The correctness of Elim is given by Theorem 1. The proof of Theorem 1 relies on monotonicity of logical operations and quantifiers in negation normal form of a formula. (Proofs for the theorems stated here can be found in [34]).

Theorem 1 (Soundness). *The algorithm* Elim *is sound: if* $I \wedge D \models \mathsf{Elim}(G)$*, then* $I \wedge D \models G$*. What is more,* $I \wedge D \wedge \mathsf{Elim}(G) \models G$*.*

We now analyze the classes of formulas G for which Elim is *complete*.

Definition 2 (Completeness). *We say that* Elim *is complete for* (D, G) *iff* $I \wedge D \models G$ *implies* $I \wedge D \models \mathsf{Elim}(G)$*.*

Note that we cannot hope to achieve completeness for arbitrary constraints D. Indeed, if we let $D \equiv \mathsf{true}$, then D imposes no constraint whatsoever on the derived fields, and reasoning about the derived fields becomes reasoning about uninterpreted function symbols, that is, reasoning in unconstrained predicate logic. Such reasoning is undecidable not only for monadic second-order logic, but also for much weaker fragments of first-order logic [7]. Despite these general observations, we have identified two cases important in practice for which Elim is complete (Theorem 2 and Theorem 3).

Theorem 2 expresses the fact that, in the case where all field constraints are deterministic, Elim is complete (and then it reduces to previous approaches [11, 26] that are restricted to the deterministic case). The proof of Theorem 2 uses the assumption that F is total and functional to conclude $\forall x\ y.\ \mathsf{FC}_f(x, y) \ \rightarrow\ f(x) = y$, and then uses an inductive argument similar to the proof of Theorem 1.

Theorem 2 (Completeness for deterministic fields). Elim *is complete for* (D, G) *when each field constraint in* D *is deterministic. Moreover,* $I \wedge D \wedge G \models \mathsf{Elim}(G)$*.*

We next turn to completeness in the cases that admit non-determinism of derived fields. Theorem 3 states that our algorithm is complete for derived fields introduced by the weakest precondition operator to a class of postconditions that includes field constraints. This result is important in practice: a previous, incomplete, version of our elimination algorithm was not able to verify the skip list example in Section 2.2. To formalize our completeness result, we introduce two classes of well-behaved formulas: *nice formulas* and *pretty nice formulas*.

Definition 3 (Nice formulas). *A formula* G *is a* nice formula *if each occurrence of each derived field* f *in* G *is of the form* $f(t)$*, where* $t \in \mathsf{Ground}(G)$*.*

Nice formulas generalize the notion of quantifier-free formulas by disallowing quantifiers only for variables that are used as arguments to derived fields. We can show that the elimination of derived fields from nice formulas is complete. The intuition behind this result is that if $I \wedge D \models G$, then for the choice of y_i such that $\mathsf{FC}_f(x_i, y_i)$ we can find an interpretation of the function symbol f such that $f(x_i) = y_i$, and $I \wedge D$ holds, so G holds as well, and $\mathsf{Elim}(G)$ evaluates to the same truth value as G.

Definition 4 (Pretty nice formulas). *The set of* pretty nice formulas *is defined inductively by 1) a nice formula is pretty nice; 2) if* G_1 *and* G_2 *are pretty nice, then* $G_1 \wedge G_2$ *and* $G_1 \vee G_2$ *are pretty nice; 3) if* G *is pretty nice and* x *is a first-order variable, then* $\forall x.G$ *is pretty nice.*

Pretty nice formulas therefore additionally admit universal quantification over arguments of derived fields. We define the function skolem, which strips (top-level) universal quantifiers, as follows: 1) $\mathsf{skolem}(G_1\ op\ G_2) = \mathsf{skolem}(G_1)\ op\ \mathsf{skolem}(G_2)$

where $op \in \{\vee, \wedge\}$; 2) $\mathsf{skolem}(\forall x.G) = G$; and 3) $\mathsf{skolem}(G) = G$, otherwise. Note that pretty nice formulas are closed under wlp (up to formula equivalence); the closure property follows from the conjunctivity of the weakest precondition operator.

$x \in \mathsf{Var}$ – program variables $\qquad f \in \mathsf{Fld}$ – pointer fields
$e \in \mathsf{Exp} ::= x \mid e.f$ $\qquad F$ – quantifier free formula
$c \in \mathsf{Com} ::= e_1 := e_2 \mid \mathsf{assume}(F) \mid \mathsf{assert}(F)$
$\quad \mid \mathsf{havoc}(x)$ (non-deterministic assignment to x)
$\quad \mid c_1 \, ; c_2 \mid c_1 \,\square\, c_2$ (sequential composition and non-deterministic choice)

Fig. 10. Loop-free statements of a guarded command language (see e.g. [1])

Theorem 3 implies that Elim is a complete technique for checking preservation (over straight-line code) of field constraints, even if they are conjoined with additional pretty nice formulas. Elimination is also complete for data structure operations with loops as long as the necessary loop invariants are pretty nice.

Theorem 3 (Completeness for preservation of field constraints). *Let G be a pretty nice formula, D a conjunction of field constraints, and c a guarded command (Figure 10). Then*

$$I \wedge D \models \mathsf{wlp}(c, G \wedge D) \quad \textit{iff} \quad I \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))) \ .$$

Example 1. The example in Figure 11 demonstrates the elimination of derived fields using algorithm Elim. It is inspired by the skip list module from Section 2.

$$\mathsf{D}_{nextSub} \equiv \forall v_1 \, v_2.\ nextSub(v_1) = v_2 \rightarrow next^+(v_1, v_2)$$

$$\begin{aligned} G &\equiv \mathsf{wlp}((e.nextSub := root.nextSub \, ; \ e.next := root), \mathsf{D}_{nextSub}) \\ &\equiv \forall v_1 \, v_2.\ nextSub[e := nextSub(root)](v_1) = v_2 \rightarrow (next[e := root])^+(v_1, v_2) \end{aligned}$$

$$\begin{aligned} G' \equiv\ & \mathsf{skolem}(\mathsf{Elim}(G)) \equiv \\ & x_1 = root \rightarrow next^+(x_1, y_1) \rightarrow \\ & \quad x_2 = v_1 \rightarrow next^+[e := y_1](x_2, y_2) \wedge (x_2 = x_1 \rightarrow y_2 = y_1) \rightarrow \\ & \qquad y_2 = v_2 \rightarrow (next[e := root])^+(v_1, v_2) \end{aligned}$$

Fig. 11. Elimination of derived fields from a pretty nice formula. The notation $next^+$ denotes the irreflexive transitive closure of predicate $next(x) = y$.

The formula G expresses the preservation of field constraint $\mathsf{D}_{nextSub}$ for updates of fields $next$ and $nextSub$ that insert e in front of $root$. The formula G is valid under the assumption that $\forall x.\ next(x) \neq e$. Elim first replaces the inner occurrence $nextSub(root)$ and then the outer occurrence of $nextSub$. Theorem 3 implies that the resulting formula $\mathsf{skolem}(\mathsf{Elim}(G))$ is valid under the same assumptions as the original formula G.

Limits of completeness. In our implementation, we have successfully used Elim in the context of MSOL, where we encode transitive closure using second-order quantification. Unfortunately, formulas that contain transitive closure of derived fields are often not pretty nice, leading to false alarms after the application of Elim. This behavior is to be expected due to the undecidability of transitive closure logics over general graphs [10]. On the other hand, unlike approaches based on axiomatizations of transitive closure in first-order logic, our use of MSOL enables complete reasoning about reachability over the backbone fields. It is therefore useful to be able to consider a field as part of a backbone whenever possible. For this purpose, it can be helpful to verify conjunctions of constraints using different backbones for different conjuncts.

Verifying conjunctions of constraints. In our skip list example, the field `nextSub` forms an acyclic (sub-)list. It is therefore possible to verify the conjunction of constraints independently, with `nextSub` a derived field in the first conjunct (as in Section 2.2) but a backbone field in the second conjunct. Therefore, although the reasoning about transitive closure is incomplete in the first conjunct, it is complete in the second conjunct.

Verifying programs with loop invariants. The technique described so far supports the following approach for verifying programs annotated with loop invariants:

1. generate verification conditions using loop invariants, pre-, and postconditions;
2. eliminate derived fields from verification conditions using Elim (and skolem);
3. decide the resulting formula using a decision procedure such as MONA [13].

Field constraints specific to program points. Our completeness results also apply when, instead of having one global field constraint, we introduce different field constraints for each program point. This allows the developer to refine data structure invariants with information specific to particular program points.

Field constraint analysis and loop invariant inference. Field constraint analysis is not limited to verification in the presence of loop invariants. In combination with abstract interpretation [3] it can be used to infer loop invariants automatically. Our implementation combines field constraint analysis with symbolic shape analysis based on Boolean heaps [33, 29] to infer loop invariants that are disjunctions of universally quantified Boolean combinations of unary predicates over heap objects.

Symbolic shape analysis casts the idea of three-valued shape analysis [32] in the framework of predicate abstraction. It uses the machinery of predicate abstraction to automatically construct the abstract post operator; this construction proceeds solely by deductive reasoning. The computation of the abstraction amounts to checking validity of entailments that are of the form: $\Gamma \wedge C \quad \rightarrow \quad \mathsf{wlp}(c, p)$. Here Γ is an over-approximation of the reachable states, C is a conjunction of abstraction predicates and p is a single abstraction predicate. We use field constraint analysis to check validity of these formulas by augmenting them with the appropriate simulation invariant I and field constraints D that specify the data structure invariants we want to preserve: $I \wedge D \wedge \Gamma \wedge C \quad \rightarrow \quad \mathsf{wlp}(c, p)$. The only problem arises from the fact that these additional invariants may be temporarily violated during program execution. To ensure applicability of the analysis, we abstract complete loop free paths in the control flow

graph of the program at once. This means that we only require that simulation invariants and field constraints are valid at loop cut points; effectively, these invariants are implicit conjuncts in each loop invariant. This approach supports the programming model where violations of invariants are confined to the interior of basic blocks [26].

Amortizing invariant checking in loop invariant inference. A straightforward approach for combining field constraint analysis with abstract interpretation would do a well-formedness check for global invariants and field constraints at every step of the fixed-point computation, invoking a decision procedure at each iteration step. The following insight allows us to use a single well-formedness check per basic block: *the loop invariant synthesized in the presence of well-formedness check is identical to the loop invariant synthesized by ignoring the well-formedness check*. We therefore speculatively compute the abstraction of the system under the assumption that both the simulation invariant and the field constraints are preserved. After the least fixed-point $\mathsf{lfp}^{\#}$ of the abstract system has been computed, we generate for every loop free path c with start point ℓ_c a verification condition: $I \wedge D \wedge \mathsf{lfp}^{\#}_{\ell_c} \rightarrow \mathsf{wlp}(c, I \wedge D)$ where $\mathsf{lfp}^{\#}_{\ell_c}$ is the projection of $\mathsf{lfp}^{\#}$ to program location ℓ_c. We then use again our Elim algorithm to eliminate derived fields and check the validity of these verification conditions. If they are all valid then the analysis is sound and the data structure invariants are preserved. Note that this approach succeeds whenever the straightforward approach would have succeeded, so it improves analysis performance without degrading precision. Moreover, when the analysis detects an error, it repeats the fixed-point computation with the simple approach to obtain an indication of the error trace.

4 Deployment as Modular Analysis Plugin

We have implemented our field constraint analysis and deployed it as the *Bohne* and *Bohne decaf* [2] analysis plugins of our Hob framework [21, 16]. We have successfully verified singly-linked lists, doubly-linked lists with and without iterators and header nodes, insertion into a tree with parent pointers, two-level skip lists (Section 2.2), and our students example from Section 2. When the developer supplies loop invariants, these benchmarks, including skip list, verify in 1.7 seconds (for the doubly-linked list) to 8 seconds (for insertion into a tree). Bohne automatically infers loop invariants for insertion and lookup in the two-level skip list in 30 minutes total. We believe the running time for loop invariant inference can be reduced using ideas such as lazy predicate abstraction [8].

Because we have integrated Bohne into the Hob framework, we were able to verify just the parts of programs which require the power of field constraint analysis with the Bohne plugin, while using less detailed analyses for the remainder of the program. We have used the list data structures verified with Bohne as modules of larger examples, such as the 900-line Minesweeper benchmark and the 1200-line web server benchmark. Hob's pluggable analysis approach allowed us to use the typestate plugin [20] and loop invariant inference techniques to efficiently verify client code, while reserving shape analysis for the container data structures.

[2] Bohne decaf is a simpler version of Bohne that does not do loop invariant inference.

5 Further Related Work

We are not aware of any previous work that provides completeness guarantees for analyzing tree-like data structures with non-deterministic cross-cutting fields for expressive constraints such as MSOL. TVLA [32, 24] was initially designed as an analysis framework with user-supplied transfer functions; subsequent work addresses synthesis of transfer functions using finite differencing [31], which is not guaranteed to be complete. Decision procedures [25, 18] are effective at reasoning about local properties, but are not complete for reasoning about reachability. Promising, although still incomplete, approaches include [23] as well as [28, 19]. Some reachability properties can be reduced to first-order properties using hints in the form of ghost fields [15, 25]. Completeness of analysis can be achieved by representing loop invariants or candidate loop invariants by formulas in a logic that supports transitive closure [26, 36, 17, 35, 37, 33, 29]. These approaches treat decision procedure as a black box and, when applied to MSOL, inherit the limitations of structure simulation [11]. Our work can be viewed as a technique for lifting existing decision procedures into decision procedures that are applicable to a larger class of structures. Therefore, it can be incorporated into all of these previous approaches.

6 Conclusion

Historically, the primary challenge in shape analysis was seen to be dealing effectively with the extremely precise and detailed consistency properties that characterize many (but by no means all) data structures. Perhaps for this reason, many formalisms were built on logics that supported *only* data structures with very precisely defined referencing relationships. This paper presents an analysis that supports both the extreme precision of previous approaches and the controlled reduction in the precision required to support a more general class of data structures whose referencing relationships may be random, depend on the history of the data structure, or vary for some other reason that places the referencing relationships inherently beyond the ability of previous logics and analyses to characterize. We have deployed this analysis in the context of the Hob program analysis and verification system; our results show that it is effective at 1) analyzing individual data structures to 2) verify interfaces that allow other, more scalable analyses to verify larger-grain data structure consistency properties whose scope spans larger regions of the program.

In a broader context, we view our result as taking an important step towards the practical application of shape analysis. By supporting data structures whose backbone functionally determines the referencing relationships as well as data structures with inherently less structured referencing relationships, it promises to be able to successfully analyze the broad range of data structures that arise in practice. Its integration within the Hob program analysis and verification framework shows how to leverage this analysis capability to obtain more scalable analyses that build on the results of shape analysis to verify important properties that involve larger regions of the program. Ideally, this research will significantly increase our ability to effectively deploy shape analysis and other subsequently enabled analyses on important programs of interest to the practicing software engineer.

Acknowledgements. We thank Patrick Maier, Alexandru Salcianu, and anonymous referees for comments on the presentation of the paper.

References

1. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer-Verlag, 1998.
2. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005.
3. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
4. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI'03*, volume 2575 of *LNCS*, pages 310–323, 2003.
5. P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.
6. R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.
7. E. Grädel. Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In *Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged*, 2003.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
9. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
10. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.
11. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. Verification via structure simulation. In *CAV*, pages 281–294, 2004.
12. J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997.
13. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.
14. N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.
15. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th POPL*, 2002.
16. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Implications of a data structure consistency checking system. In *Int. conf. on Verified Software: Theories, Tools, Experiments (VSTTE, IFIP Working Group 2.3 Conference)*, Zürich, October 2005.
17. V. Kuncak and M. Rinard. Boolean algebra of shape analysis constraints. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.
18. V. Kuncak and M. Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.
19. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006.
20. P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.
21. P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005.
22. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

23. T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005.
24. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.
25. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005.
26. A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.
27. S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981.
28. G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.
29. A. Podelski and T. Wies. Boolean heaps. In *SAS*, 2005.
30. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.
31. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003.
32. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
33. T. Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, Sep 2004.
34. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. Technical Report MIT-CSAIL-TR-2005-072, MIT-LCS-TR-1010, MIT CSAIL, November 2005.
35. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004.
36. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 2005. (to appear).
37. G. Yorsh, A. Skidanov, T. Reps, and M. Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005.

A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety*

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula

University of California, Berkeley, California, USA
{bec, adamc, necula}@cs.berkeley.edu

Abstract. A *certified program analysis* is an analysis whose implementation is accompanied by a checkable proof of soundness. We present a framework whose purpose is to simplify the development of certified program analyses without compromising the run-time efficiency of the analyses. At the core of the framework is a novel technique for automatically extracting Coq proof-assistant specifications from ML implementations of program analyses, while preserving to a large extent the structure of the implementation. We show that this framework allows developers of mobile code to provide to the code receivers untrusted code verifiers in the form of certified program analyses. We demonstrate efficient implementations in this framework of bytecode verification, typed assembly language, and proof-carrying code.

1 Introduction

When static analysis or verification tools are used for validating safety-critical code [6], it becomes important to consider the question of whether the results of the analyses are trustworthy [22, 3]. This question is becoming more and more difficult to answer as both the analysis algorithms and their implementations are becoming increasingly complex in order to improve precision, performance, and scalability. We describe a framework whose goal is to assist the developers of program analyses in producing formal proofs that the implementations and algorithms used are sound with respect to a concrete semantics of the code. We call such analyses *certified* since they come with machine-checkable proofs of their soundness. We also seek soundness assurances that are *foundational*, that is, that avoid assumptions or trust relationships that don't seem fundamental to the objectives of users. Our contributions deal with making the development of such analyses more practical, with particular emphasis on not sacrificing the efficiency of the analysis in the process.

The strong soundness guarantees given by certified program analyzers and verifiers are important when the potential cost of wrong results is significant.

* This research was supported in part by NSF Grants CCR-0326577, CCF-0524784, and CCR-00225610; an NSF Graduate Fellowship; and an NDSEG Fellowship. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 174–189, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Moreover, the ability to check independently that the implementation of the analysis is sound allows us to construct a mobile-code receiver that allows untrusted parties to provide the code verifier. The code verifier is presented as a certified program analysis whose proof of soundness entails soundness of code verification.

The main contributions of the framework we propose are the following:

- We describe a methodology for translating automatically implementations of analyses written in a general-purpose language (currently, ML) into models and specifications for a proof assistant (currently, Coq). Specifically, we show how to handle those aspects of a general-purpose language that do not translate directly to the well-founded logic used by the proof assistant, such as side-effects and non-primitive recursive functions. We use the framework of abstract interpretation [12] to derive the soundness theorems that must be proved for each certified analysis.
- We show a design for a flexible and efficient mobile-code verification protocol, in which the untrusted code producer has complete freedom in the safety mechanisms and compilation strategies used for mobile code, as long as it can provide a code verifier in the form of a certified analysis, whose proof of soundness witnesses that the analysis enforces the desired code-receiver safety policy.

In the next section, we describe our program analysis framework and introduce an example analyzer. Then, in Sect. 3, we present our technique for specification extraction from code written in a general-purpose language. We then discuss the program analyzer certification process in Sect. 4. In Sect. 5, we present an application of certified program analysis to mobile code safety and highlight its advantages and then describe how to implement in this architecture (foundational) typed assembly language, Java bytecode verification, and proof-carrying code in Sect. 6. Finally, we survey related work (Sect. 7) and conclude (Sect. 8).

2 The Certified Program Analysis Framework

In order to certify a program analysis, one might consider proving directly the soundness of the implementation of the analysis. This is possible in our framework, but we expect that an alternative strategy is often simpler. For each analysis to be certified, we write a certifier that runs after the analysis and checks its results. Then, we prove the soundness of the certifier. This approach has several important advantages. Often the certifier is simpler than the analysis itself. For example, it does not need to iterate more than once over each instruction, and it does not need all the complicated heuristics that the analysis itself might use to speed up the convergence to a fixpoint. Thus, we expect the certifier is easier to prove sound than the analysis itself. The biggest benefit, however, is that we can use an existing implementation of a program analysis as a black box, even if it is written in a language that we are not ready to analyze formally, and even if the analysis algorithm does not fit perfectly with the

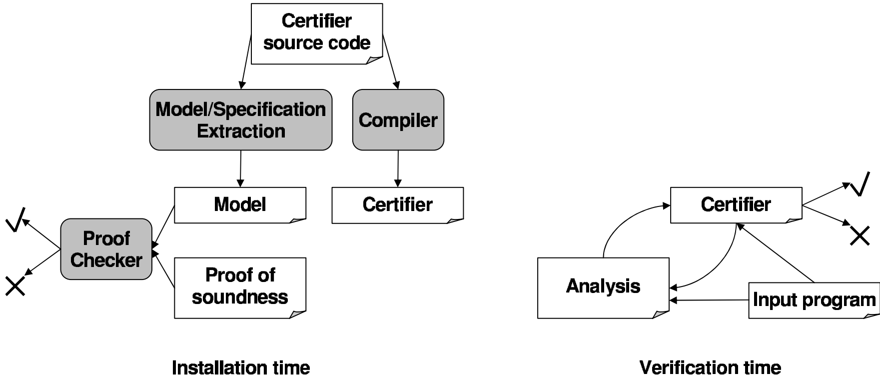


Fig. 1. Our certified verifier architecture with the trusted code base shaded

formalism desired for the certification and its soundness proofs. As an extreme example, the analysis itself might contain a model checker, while we might want to do the soundness proof using the formalism of abstract interpretation [28]. In Fig. 1, we diagram this basic architecture for the purpose of mobile-code safety. We distinguish between "installation time" activity, which occurs once per analyzer, and "verification time" activity, which occurs once per program to analyze.

We choose the theory of abstract interpretation [12] as the foundation for the soundness proofs of certifiers because of its generality and because its soundness conditions are simple and well understood. We present first the requirements for the developers of certifiers, and then in Sect. 4, we describe the soundness verification.

The core of the certifier is an untrusted custom module containing an implementation of the abstract transition relation (provided by the certifier developer). The custom module of a certifier must implement the signature given adjacently.

```
type absval
type abs = { pc : nat; a : absval }
val ainv : abs list
val astep : abs -> result
datatype result = Fail | Succ of abs list
```

The type `abs` encodes abstract states, which include a program counter and an abstract value of a type that can be chosen by the certifier developer. The value `ainv` consists of the abstract invariants. They must at a minimum include invariants for the entry points to the code and for each destination of a jump. The function `astep` implements the abstract transition relation: given an abstract state at a particular instruction, compute the set of successor states, minus the states already part of `ainv`. The transition relation may also fail, for example when the abstract state does not ensure the safe execution of the instruction. We will take advantage of this possibility to write safety checkers for mobile-code using this framework. In our implementation and in the examples in this paper, we use the ML language for implementing custom certifiers.

In order to execute such certifiers, the framework provides a trusted engine shown in Fig. 2. The main entry point is the function `top`, invoked with a list of program counters that have been processed and a list of abstract states still to process. Termination is ensured using two mechanisms: each invocation of the untrusted `astep` is guarded by a timeout, and each program counter is processed at most once. We use a timeout as a simple alternative to proving termination of `astep`. A successful run of the code shown in Fig. 2 is intended to certify that all of the abstract states given by `ainv` (i.e., the properties that we are verifying for a program) are invariant, and that the `astep` function succeeds on all reachable instructions. We take advantage of this latter property to write untrusted code verifiers in this framework (Sect. 5). We discuss these guarantees more precisely in Sect. 4.

```
fun applyWithTimeout (f: 'a -> 'b, x: 'a) : 'b = ...
fun top (DonePC: nat list, ToDo: abs list) : bool =
 case ToDo of
   nil => true
 | a :: rest =>
      if List.member(a.pc, DonePC) then false else
      (case applyWithTimeout(astep, a) of
         Fail => false
       | Succ as => top (a.pc :: DonePC, as @ ToDo))
in
  top (nil, ainv)
```

Fig. 2. The trusted top-level analysis engine. The infix operators :: and @ are list cons and append, respectively.

Example: Java Bytecode Verifier. Now we introduce an example program analyzer that requires the expressivity of a general-purpose programming language and highlights the challenges in specification extraction. In particular, we consider a certifier in the style of the Java bytecode verifier, but operating on a simple assembly language instead of bytecodes. Fig. 3 presents a fragment of this custom verifier. The abstract value is a partial map from registers to class names, with a missing entry denoting an uninitialized register.[1]

In the `astep` function, we show only the case of the memory write instruction. The framework provides the `sel` accessor function for partial maps, the instruction decoder `instrAt`, the partial function `fieldOf` that returns the type of a field at a certain offset, and the partial function `super` that returns the super class of a class. This case succeeds only if the destination address is of the form `rdest` + `n`, with register `rdest` pointing to an object of class `cdest` that has at offset `n` a field of type `c'`, which must be a super class of the type of register `rsrc`.

We omit the code for `calculatePreconditions`, a function that obtains some preconditions from the meta-data packaged with the `.class` files, and then uses an iterative fixed-point algorithm to find a good typing precondition for each program label. Each such precondition should be satisfied any time control reaches its label. This kind of algorithm is standard and well studied, in

[1] In the actual implementation, registers that hold code pointers (e.g., return addresses, or dynamic dispatch addresses) are assigned types that specify the abstract state expected by the destination code block.

```
type absval = (reg, class) partialmap
type abs = { pc : nat; a : absval }

fun subClass (c1 : class, c2 : class) = ...
fun calculatePreconditions () : abs list = ...
val ainv: abs list = calculatePreconditions ()

fun astep (a : abs) : result =
  case instrAt(a.pc) of
    Write(rdest + n, rsrc) =>
      (case (sel(a.a, rdest), sel(a.a, rsrc)) of
         (SOME cdest, SOME csrc) =>
           (case fieldOf(cdest, n) of
              SOME cdest' => if subClass(csrc, cdest') then
                                 Succ [ { a = a.a, pc = a.pc + 1 } ]
                             else Fail
            | _ => Fail)
       | _ => Fail)
  | ...
```

Fig. 3. Skeleton of a verifier in the style of the Java bytecode verifier

the context of the Java Bytecode Verifier and elsewhere, so we omit the details here. Most importantly, we will not need to reason formally about the correctness of this algorithm.

3 Specification Extraction

To obtain certified program analyses, we need a methodology for bridging the gap between an implementation of the analysis and a specification that is suitable for use in a proof assistant. An attractive technique is to start with the specification and its proof, and then use *program extraction* supported by proof assistants such as Coq or Isabelle [29] to obtain the implementation. This strategy is very proof-centric and while it does yield a sound implementation, it makes it hard to control non-soundness related aspects of the code, such as efficiency, instrumentation for debugging, or interaction with external libraries.

Yet another alternative is based on *verification conditions* [15, 17], where each function is first annotated with a pre- and postcondition, and the entire program is compiled into a single formula whose validity implies that the program satisfies its specification. Such formulas can make good inputs to automated deduction tools, but they are usually quite confusing to a human prover. They lose much of the structure of the original program. Plus, in our experience, most auxiliary functions in a program analyzer do good jobs of serving as their own specifications (e.g., the `subClass` function).

Since it is inevitable that proving soundness will be sufficiently complicated to require human guidance, we seek an approach that maintains as close of a correspondence between the implementation and its model as possible. For non-

```
fun subClass (depth : nat, c1 : class, c2 : class) : bool option =
  c1 = c2 orelse
  (case super c1 of NONE => SOME false
   | SOME sup => if depth = 0 then NONE else subClass'(depth-1, sup,c2))
```

Fig. 4. Translation of the `subClass` function. The boxed elements are added by our translation.

recursive purely functional programs, we can easily achieve the ideal, as the implementation can reasonably function as its own model in a suitable logic, such as that of the Coq proof assistant. This suggests that we need a way to handle imperative features, and a method for dealing with non-primitive recursive functions. In the remainder of this section, we give an overview of our approach. More detail can be found in the companion technical report [9].

Handling Recursion. We expect that all invocations of the recursive functions used during certification terminate, although it may be inconvenient to write all functions in primitive recursive form, as required by Coq. In our framework, we force termination of all function invocations using timeouts. This means that for each successful run (i.e., one that does not time out) there is a bound on the call-stack depth. We use this observation to make all functions primitive recursive on the call-stack depth. When we translate a function definition, we add an explicit argument `depth` that is checked and decremented at each function call. Fig. 4 shows the result of translating a typical implementation of the `subClass` function for our running example. The boxed elements are added by the translation. Note that in order to be able to signal a timeout, the return type of the function is an `option` type. Coq will accept this function because it can check syntactically that it is primitive recursive in the `depth` argument.

This translation preserves any *partial correctness* property of the code. For example, if we can prove about the specification that any invocation of `subClass` that yields `SOME true` implies that two classes are in a subclass relationship, then the same property holds for the original code whenever it terminates with the value `true`.

Handling Imperative Features. The function `calculatePreconditions` from Fig. 3 uses I/O operations to read and decode the basic block invariants from the `.class` file (as in the KVM [30] version of Java), or must use an intraprocedural fixed-point computation to deduce the basic block preconditions from the method start precondition (as for standard `.class` files). In any case, this function most likely uses a significant number of imperative constructs or even external libraries. This example demonstrates a situation when the result of complex computations is used only as a *hint*, whose exact value is not important for *soundness* but only for completeness. We believe that this is often the case when writing certifiers, which suggests that a monadic [31] style of translation would unnecessarily complicate the resulting specification.

For such situations we propose a cheaper translation scheme that abstracts soundly the result of side-effecting operations. We describe this scheme informally,

```
fun readu16 (s: callstate,  buff: int array, idx: int) : int =
  256 * (freshread1 s) + (freshread2 s)

fun readu32 (s: callstate,  buff: int array, idx: int) : int =
  65536 * readu16(freshstate3 s,buff,i) + readu16(freshstate4 s,buff,i+2)
```

Fig. 5. Translation of a function for reading a 16-bit and 32-bit big-endian numbers from a class file. Original body of `readu16` before translation is $256 * \mathtt{buff}[\mathtt{i}] + \mathtt{buff}[\mathtt{i} + 1]$.

by means of an example of functions that read from a Java `.class` file 16-bit and 32-bit numbers, respectively, written in big-endian notation, shown in Fig. 5. Each update to mutable state is ignored. Each syntactic occurrence of a mutable-state access is replaced with a fresh abstract function (e.g., `freshread1`) whose argument is an abstraction of the call-stack state. The call-stack argument is needed to ensure that no relationship can be deduced between recursive invocations of the same syntactic state access. Each function whose body reads mutable state, or calls functions that read mutable state, gets a new parameter `s` that is the abstraction of the call-stack state. Whenever such a function calls another function that needs a call-stack argument, it uses a fresh transformer (e.g., `freshstate3`) to produce the new actual state argument.

This abstraction is sound in the sense that it ensures that nothing can be proved about results of mutable state accesses, and thus any property that we can prove about this abstraction also holds for the actual implementation. If we did not have the call-stack argument, one could prove that each invocation of the `readu16` function produces the same result, and thus all results of the `readu32` are multiple of 65,537. This latter example also shows why we cannot use the `depth` argument as an abstraction of the call-stack state.

Note that our use of “state” differs from the well-known “explicit state-passing style” in functional programming, where state is used literally to track all mutable aspects of the execution environment. That translation style requires that each function that updates the state not only take an input state but also produce an output state that must be passed to the next statement. In our translation scheme states are only passed down to callers, and the result type of a function does not change.

The cost for the simplicity of this translation is a loss of completeness. We are not interested in preserving all the semantics of input programs. Based on our conjecture that we can refactor programs so that their soundness arguments do not depend on imperative parts, we can get away with a looser translation. In particular, we want to be able to prove properties of the input by proving properties of the translation. We do not need the opposite inclusion to hold.

Soundness of the Specification Extraction. We argue here informally the soundness of the specification extraction for mutable state. In our implementation, the soundness of the code that implements the extraction procedure is assumed. We leave for future work the investigation of ways to relax this assumption. First,

we observe that each syntactic occurrence of a function call has its own unique `freshstate` transformer. This means that, in an execution trace of the specification, each function call has an actual state argument that is obtained by a unique sequence of applications of `freshstate` transformers to the initial state. Furthermore, in any such function invocation all the syntactic occurrences of a mutable state read use unique `freshread` access functions, applied to unique values of the state parameter. This means that in any execution trace of the specification, each state read value is abstracted as a unique combination of `freshread` and `freshstate` functions. This, in turn, means that for any actual execution trace of the *original program*, there is a definition of the `freshread` and `freshstate` parameters that yields the same results as the actual reads. Since all the `freshread` and `freshstate` transformers are left abstract in the specification, any proof about the specification works with any model for the transformers, and thus applies to any execution trace of the original program. A complete formal proof is found in the companion technical report [9].

4 Soundness Certification

We use the techniques described in the previous section to convert the ML data type `abs` to a description of the abstract domain $\mathcal{A}$ in the logic of the proof-assistant. Similarly, we convert the `ainv` value into a set $\mathcal{A}_I \subseteq \mathcal{A}$. Finally, we model the transition function `astep` as an abstract transition relation $\rightsquigarrow \subseteq \mathcal{A} \times 2^{\mathcal{A}}$ such that $a \rightsquigarrow A$ whenever $\texttt{astep}(a) = \texttt{Succ}\ A$. We will abuse notation slightly and identify sets and lists where convenient.

We prove soundness of the abstract transition relation with respect to a concrete transition relation. Let $(\mathcal{C}, \mathcal{C}_0, \mapsto)$ be a *transition system* for the concrete machine. In particular, $\mathcal{C}$ is a domain of states; $\mathcal{C}_0$ is the set of allowable initial states; and $\mapsto$ is a one-step transition relation. These elements are provided in the proof-assistant logic and are trusted. We build whatever safety policy interests us into $\mapsto$ in the usual way; we disallow transitions that would violate the policy, so that errors are modeled as "being stuck." This is the precise way in which one can specify the trusted *safety policy* for the certified program verifiers (Sect. 5).

To certify the soundness of the program analyzer, the certifier developer needs to provide additionally (in the form of a Coq definition) a *soundness relation* $\simeq\ \subseteq \mathcal{C} \times \mathcal{A}$ (written as σ in [13]), such that $c \simeq a$ holds if the abstract state a is a sound abstraction of the concrete state c. To demonstrate $\simeq$ is indeed sound, the author also provides proofs (in Coq) for the following standard, local soundness properties of abstract interpretations and bi-simulations.

Property 1 (Initialization). For every $c \in \mathcal{C}_0$, there exists $a \in \mathcal{A}_I$ such that $c \simeq a$.

The initialization property assures us that the abstract interpretation includes an appropriate invariant for every possible concrete initial state.

Property 2 (Progress). For every $c \in \mathcal{C}$ and $a \in \mathcal{A}$ such that $c \simeq a$, if there exists $A' \subseteq \mathcal{A}$ such that $a \rightsquigarrow A'$, then there exists $c' \in \mathcal{C}$ such that $c \mapsto c'$.

Progress guarantees that, whenever an abstract state is not stuck, any corresponding concrete states are also not stuck.

Property 3 (Preservation). For every $c \in \mathcal{C}$ and $a \in \mathcal{A}$ such that $c \simeq a$, if there exists $A' \subseteq \mathcal{A}$ such that $a \rightsquigarrow A'$, then for every $c' \in \mathcal{C}$ such that $c \mapsto c'$ there exists $a' \in (A' \cup \mathcal{A}_I)$ such that $c' \simeq a'$.

Preservation guarantees that, for every step made by the concrete machine, the resulting concrete state matches one of the successor states of the abstract machine. Preservation is only required when the abstract machine does not reject the program. This allows the abstract machine to reject some safe programs, if it so desires. It is important to notice that, in order to ensure termination, the `astep` function (and thus the $\rightsquigarrow$ relation) only returns those successor abstract states that are not already part of the initial abstract states `ainv`. To account for this aspect, we use $\mathcal{A}_I$ in the preservation theorem.

Together, these properties imply the global soundness of the certifier that implements this abstract interpretation [12], stated as following:

Theorem 1 (Certification soundness). *For any concrete state $c \in \mathcal{C}$ reachable from an initial state in $\mathcal{C}_0$, the concrete machine can make further progress. Also, if c has the same program counter as a state $a \in \mathcal{A}_I$, then $c \simeq a$.*

In the technical report [9], we give an idea how these obligations are met in practice by sketching how the proof goes for the example of the Java bytecode verifier shown in Fig. 3.

5 Applications to Mobile-Code Safety

Language-based security mechanisms have gained acceptance for enforcing basic but essential safety properties, such as memory and type safety, for untrusted mobile code. The most widely deployed solution for mobile code safety is bytecode verification, as in the Java Virtual Machine (JVM) [25] or the Microsoft Common Intermediate Language (MS-CIL) [18]. A bytecode verifier uses a form of abstract interpretation to track the types of machine registers, and to enforce memory and type safety. The main limitation of this approach is that we must trust the soundness of the bytecode verifier. In turn, this means that we cannot easily change the verifier and its enforcement mechanism. This effectively forces the clients of a code receiver to use a fixed type system and often even a fixed source language for mobile code. Programs written in other source languages can be compiled into the trusted intermediate language but often in unnatural ways with a loss of expressiveness and performance [4, 19, 7].

A good example is the MS-CIL language, which is expressive enough to be the target of compilers for C#, C and C++. Compilers for C# produce intermediate code that can be verified, while compilers for C and C++ use intermediate language instructions that are always rejected by the built-in bytecode verifier. In this latter case, the code may be accepted if the producer of the code can provide an explicit proof that the code obeys the required safety policy and the code receiver uses proof-carrying code [1, 20, 27].

Existing work on proof-carrying code (PCC) attests to its versatility, but often fails to address the essential issue of how the proof objects are obtained. In the Touchstone system [11], proofs are generated by a special theorem prover with detailed knowledge about Java object layout and compilation strategies. The Foundational PCC work [1, 20] eliminates the need to hard-code and trust all such knowledge, but does so at the cost of increasing many times the proof generation burden. Both these systems also incur the cost of transmitting proofs. The Open Verifier project [10] proposes to send with the code not per-program proofs but proof generators to be run at the code receiver end for each incoming program. The generated proofs are then checked by a trusted proof checker, as in a standard PCC setup.

Using certified program analyses we can further improve this process. The producer of the mobile code writes a safety-policy verifier customized for the exact compilation strategy and safety reasoning used in the generation of the mobile code. This verifier can be written in the form of a certified program analysis, whose abstract transition fails whenever it cannot verify the safety of an instruction. For example, we discuss in Sect. 6 cases when the program analysis is a typed assembly language checker, a bytecode verifier, or an actual PCC verification engine relying on annotations accompanying the mobile code.

The key element is the soundness proof that accompanies an analysis, which can be checked automatically. At verification time, the now-trusted program analyzer is used to validate the code, with no need to manipulate explicit proof objects. This simplifies the writing of the validator (as compared with the proof-generating theorem prover of Touchstone, or the Open Verifier). We also show in Sect. 6 that this reduces the validation time by more than an order of magnitude.

We point out here that the soundness proof is with respect to the trusted concrete semantics. By adding additional safety checks in the concrete semantics (for instance, the logical equivalents of dynamic checks that would enforce a desired safety policy), the code receiver can construct customized safety policies.

6 Case Studies

In this section, we present case studies of applying certified program analyzers to mobile code security. We describe experience with verifiers for typed assembly language, Java bytecode, and proof-carrying code.

We have developed a prototype implementation of the certified program analysis infrastructure. The concrete language to be analyzed is the Intel x86 assembly language. The specification extractor is built on top of the front-end of the OCaml compiler, and it supports a large fragment of the ML language. The most notable features not supported are the object-oriented features. In addition to the 3000-line extractor, the trusted computing base includes the whole OCaml compiler and the Coq proof checker, neither of which is designed to be foundationally small. However, our focus here has been on exploring the ease of use and run-time efficiency of our approach. We leave minimizing the trusted base for future work.

Typed Assembly Language. Our first realistic use of this framework involved Typed Assembly Language. In particular, we developed and proved correct a verifier for TALx86, as provided in the first release of the TALC tools from Cornell [26]. This TAL includes several interesting features, including continuation, universal, existential, recursive, product, sum, stack, and array types. Our implementation handles all of the features used by the test cases distributed with TALC, with the exception of the modularity features, which we handle by "hand-linking" multiple-file tests into single files. TALC includes compilers to an x86 TAL from Popcorn (a safe C dialect) and mini-Scheme. We used these compilers unchanged in our case study.

We implemented a TALx86 verifier in 1500 lines of ML code. This compares favorably with the code size of the TALC type checker, which is about 6000 lines of OCaml. One of us developed our verifier over the course of two months, while simultaneously implementing the certification infrastructure. We expect that it should be possible to construct new verifiers of comparable complexity in a week's time now that the infrastructure is stable.

We also proved the local soundness properties of this implementation in 15,000 lines of Coq definitions and proof scripts. This took about a month, again interleaved with developing the trusted parts of the infrastructure. We re-used some definitions from a previous TAL formalization [10], but we didn't re-use any proofs. It's likely that we can significantly reduce the effort required for such proofs by constructing some custom proof tactics based on our experiences. We don't believe our formalization to be novel in any fundamental way. It uses ideas from previous work on foundational TAL [2, 20, 14]. The main difference is that we prove the same basic theorems about the behavior of an implementation of the type checker, instead of about the properties of inference rules. This makes the proofs slightly more cumbersome, but, as we will see, it brings significant performance improvement. As might be expected, we found and fixed many bugs in the verifier in the course of proving its soundness. This suggests that our infrastructure might be useful even if the developer is only interested in debugging his analysis.

Table 1. Average verifier running times (in seconds)

	Conv	CPV	PCC
Up to 200 (13)	0	0.01	0.07
201-999 (7)	0.01	0.02	0.24
1000 and up (6)	0.04	0.08	1.73

Table 1 presents some verification-time performance results for our implementation, as average running times for inputs with particular counts of assembly instructions. We ran a number of verifiers on the test cases provided with TALC, which used up to about 9000 assembly instructions. First, the type checker included with TALC finishes within the resolution of our timing technique for all cases, so we don't include results for it. While this type checker operates on a special typed assembly language, the results we give are all for verifying native assembly programs, with types and macro-instructions used as meta-data. As a result, we can expect that there should be some inherent slow-down, since some TAL instructions must be compiled to multiple real instructions. The experiments were

performed on an Athlon XP 3000+ with 1 GB of RAM, and times are given in seconds. We give times for "Conventional (Conv)," a thin wrapper around the TALC type checker to make it work on native assembly code; "CPV," our certified program verifier implementation; and "PCC," our TALx86 verifier implementation from previous work [10], in which explicit proof objects are checked during verification.

The results show that our CPV verifier performs comparably with the conventional verifier, for which no formal correctness proof exists. It appears our CPV verifier is within a small constant factor of the conventional verifier. This constant is likely because we use an inefficient, Lisp-like serialization format for including meta-data in the current implementation. We expect this would be replaced by a much faster binary-encoded system in a more elaborate version.

We can also see that the certified verifier performs much better than the PCC version. The difference in performance is due to the cost required to manipulate and check explicit proof objects during verification. To provide evidence that we aren't comparing against a poorly-constructed straw man, we can look to other FPCC projects. Wu, Appel, and Stump [32] give some performance results for their Prolog-based implementation of trustworthy verifiers. They only present results on input programs of up to 2000 instructions, with a running time of .206 seconds on a 2.2 GHz Pentium IV. This seems on par with our own PCC implementation. While their trusted code base is much smaller than ours, since we require trust in our specification extractor, there is hope that we can achieve a similarly small checking kernel by using techniques related to certifying compilation.

Java Bytecode Verification. We have also used our framework to implement a partial Java Bytecode Verifier (JBV) in about 600 lines of ML. It checks most of the properties that full JBVs check, mainly excluding exceptions, object initialization, and subroutines. Our implementation's structure follows closely that of our running example from Sect. 2. Its `ainv` begins by calling an OCaml function that calculates a fixed point using standard techniques. Like in our example, the precise code here doesn't matter, as the purpose of the function is to populate a hash table of function preconditions and control-flow join point invariants. With this information, our `astep` function implements the standard typing rules for JBVs.

While we have extracted complete proof obligations for the implementation, we have only begun the process of proving them. However, to make sure we are on track to an acceptable final product, we have performed some simple benchmarks against the bytecode verifier included with Blackdown Java for Linux. We downloaded a few Java-only projects from SourceForge and ran each verifier on every class in each project.

On the largest that our prototype implementation could handle, MegaMek, our verifier finishes in 5.5 seconds for checking 668,000 bytecode instructions, compared to 1 second for the traditional verifier. First, we note that both times are relatively small in an absolute sense. It probably takes a user considerably longer to download a software package than to verify it with either method.

We also see that our verifier is only a small factor away from matching the traditional approach, whose performance we know empirically that users seem willing to accept. No doubt further engineering effort could close this gap or come close to doing so.

Proof-Carrying Code. We can even implement a version of Foundational PCC in our framework: for each basic block the mobile code contains an invariant for the start of the block, and a proof that the strongest postcondition of the start invariant along the block implies the invariant for the successor block. The abstract state `abs` of the certifier consists of a predicate written in a suitable logic, intended to be the strongest postcondition at the given program point. The `ainv` is obtained by reading invariants from a data segment accompanying the mobile code.

```
fun checkProof (prf: proof) (p: pred) : bool = ...
fun astep (a: abs) : result =
 case instrAt a.pc of
   RegReg(r1, r2) => Succ [{
     pc = a.pc + 1;
     a = And(Eq(r1,r2),Exists(x,[x/r1]a.a)) }]
 | Jump l =>
     let dest = getInvar l in
     let prf = fetchProof l in
     if checkProof (prf, Imply(a.a, dest)) then
       Succ [ ]
     else Fail
```

Fig. 6. A fragment of a certifier for PCC

Fig. 6 shows a fragment of the code for `astep`, which calculates the strongest postcondition for every instruction. At a jump we fetch the invariant for the destination, a proof, and then check the proof. To prove soundness, we only need to ensure that `getInvar` returns one of the invariants that are part of `ainv`, and that the `checkProof` function is sound. More precisely, whenever the call to `checkProof` returns true, then any concrete state that satisfies `a.a` also satisfies `dest`. In particular, we do not care at all how `fetchProof` works, where it gets the proof from, whether it decrypts or decompresses it first, or whether it actually produces the proof itself. This soundness proof for `checkProof` is possible and even reasonably straightforward, since we are writing our meta-proofs in Coq's more expressive logic.

7 Related Work

Toward Certified Program Analyses. The Rhodium system developed by Lerner *et al.* [24] is the most similar with respect to the overall goal of our work—that of providing a realistic framework for certified program analyses. However, they focus on simpler compiler analysis problems whose soundness can be proved by today's automated methods. We expect that our proofs can similarly be automated when our framework is used for the kinds of analyses expressible in Rhodium-style domain specific languages.

Several systems have been developed for specifying program analyses in domain-specific languages and generating code from these specifications [23].

Again, the expressiveness of these systems is very limited compared to what is needed for standard mobile code safety problems.

In the other direction, we have the well-established body of work dealing with extracting formal verification conditions from programs annotated with specifications. Especially relevant are the Why [16] and Caduceus [17] tools, which produce Coq proof obligations as output.

There has been a good amount of work on constructing trustworthy verifiers by extracting their code from constructive proofs of soundness. Cachera *et al.* [8] extracted a data-flow analysis from a proof based on a general constraint framework. Klein and Nipkow [21] and Bertot [5] have built certified Java bytecode verifiers through program extraction/code generation from programs and proofs in Isabelle and Coq, respectively. None of these publications present any performance figures to suggest that their extracted verifiers scale to real input sizes.

Enforcing Mobile-Code Safety. As alluded to earlier, most prior work in Foundational Proof-Carrying Code has focused on the generality and expressivity of various formalisms, including the original FPCC project [2], Syntactic FPCC [20], and Foundational TALT [14]. These projects have given convincing arguments for their expressiveness, but they have not yet demonstrated a scalable implementation. Some recent research has looked into efficiency considerations in FPCC implementations, including work by Wu, Appel, and Stump [32] and our own work on the Open Verifier [10].

The architecture proposed by Wu, Appel, and Stump is fairly similar to the architecture we propose, with the restriction that verifiers must be implemented in Prolog. In essence, while we build in an abstract interpretation engine, Wu *et al.* build in a Prolog interpreter. We feel that it is important to support verifiers developed in more traditional programming languages. Also, the performance figures provided by Wu *et al.* have not yet demonstrated scalability.

Our past work on the Open Verifier has heavily influenced the design of the certified program analysis architecture. Both approaches build an abstract interpretation engine into the trusted base and allow the uploading of customized verifiers. However, the Open Verifier essentially adheres to a standard PCC architecture in that it still involves proof generation and checking for each mobile program to be verified, and it pays the usual performance price for doing this.

8 Conclusion

We have presented a strategy for simplifying the task of proving soundness not just of program analysis algorithms, but also of their implementations. We believe that starting with the implementation and extracting natural proof obligations will allow developers to fine tune non-functional aspects of the code, such as performance or debugging instrumentation.

Certified program analyses have immediate applications for developing certified program verifiers, such that even untrusted parties can customize the verification process for untrusted code. We have created a prototype implementation

and used it to demonstrate that the same infrastructure can support in a very natural way proof-carrying code, type checking, or data-flow based verification in the style of bytecode verifiers. Among these, we have completed the soundness proof of a verifier for x86 Typed Assembly Language. The performance of our certified verifier is quite on par with that of a traditional, uncertified TALx86 type checker. We believe our results here provide the first published evidence that a foundational code certification system can scale.

References

1. A. W. Appel. Foundational proof-carrying code. In *Proc. of the 16th Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. of the 27th Symposium on Principles of Programming Languages*, pages 243–253, Jan. 2000.
3. G. Barthe, P. Courtieu, G. Dufay, and S. de Sousa. Tool-assisted specification and verification of the JavaCard platform. In *Proc. of the 9th International Conference on Algebraic Methodology and Software Technology*, Sept. 2002.
4. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proc. of the International Conference on Functional Programming*, pages 129–140, June 1999.
5. Y. Bertot. Formalizing a JVML verifier for initialization in a theorem prover. In *Proc. of the 13th International Conference on Computer Aided Verification*, volume 2102 of *LNCS*, pages 14–24, July 2001.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 196–207, 2003.
7. P. Bothner. Kawa — compiling dynamic languages to the Java VM. In *Proc. of the FreeNIX Track: USENIX 1998 annual technical conference*, 1998.
8. D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. In D. A. Schmidt, editor, *Proc. of the 13th European Symposium on Programming*, volume 2986 of *LNCS*, pages 385–400, Mar. 2004.
9. B.-Y. E. Chang, A. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. Technical Report UCB ERL M05/32, University of California, Berkeley, 2005.
10. B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The Open Verifier framework for foundational verifiers. In *Proc. of the 2nd Workshop on Types in Language Design and Implementation*, Jan. 2005.
11. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. of the Conference on Programming Language Design and Implementation*, pages 95–107, May 2000.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symposium on Principles of Programming Languages*, pages 234–252, Jan. 1977.
13. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

14. K. Crary. Toward a foundational typed assembly language. In *Proc. of the 30th Symposium on Principles of Programming Languages*, pages 198–212, Jan. 2003.
15. E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
16. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
17. J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Proc. of the 6th International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 15–29, Nov. 2004.
18. A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proc. of the 28th Symposium on Principles of Programming Languages*, pages 248–260, Jan. 2001.
19. K. J. Gough and D. Corney. Evaluating the Java virtual machine as a target for languages other than Java. In *Joint Modula Languages Conference*, Sept. 2000.
20. N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. of the 17th Symposium on Logic in Computer Science*, pages 89–100, July 2002.
21. G. Klein and T. Nipkow. Verified lightweight bytecode verification. *Concurrency – practice and experience*, 13(1), 2001.
22. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 298(3):583–626, 2003.
23. J. H. E. F. Lasseter. Toolkits for the automatic construction of data flow analyzers. Technical Report CIS-TR-04-03, University of Oregon, 2003.
24. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd Symposium on Principles of Programming Languages*, pages 364–377, 2005.
25. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.
26. G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. Talc releases, 2003. URL: http://www.cs.cornell.edu/talc/releases.html.
27. G. C. Necula. Proof-carrying code. In *Proc. of the 24th Symposium on Principles of Programming Languages*, pages 106–119, Jan. 1997.
28. G. C. Necula, R. Jhala, R. Majumdar, T. A. Henzinger, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of the Conference on Computer Aided Verification*, Nov. 2002.
29. L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.
30. E. Rose. Lightweight bytecode verification. *J. Autom. Reason.*, 31(3-4):303–334, 2003.
31. P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.
32. D. Wu, A. W. Appel, and A. Stump. Foundational proof checkers with small witnesses. In *Proc. of the 5th International Conference on Principles and Practice of Declarative Programming*, pages 264–274, Aug. 2003.

Improved Algorithm Complexities for Linear Temporal Logic Model Checking of Pushdown Systems*

Katia Hristova** and Yanhong A. Liu

Computer Science Department, State University of New York,
Stony Brook, NY 11794
katia@cs.sunysb.edu

Abstract. This paper presents a novel implementation strategy for linear temporal logic (LTL) model checking of pushdown systems (PDS). The model checking problem is formulated intuitively in terms of evaluation of Datalog rules. We use a systematic and fully automated method to generate a specialized algorithm and data structures directly from the rules. The generated implementation employs an incremental approach that considers one fact at a time and uses a combination of linked and indexed data structures for facts. We provide precise time complexity for the model checking problem; it is computed automatically and directly from the rules. We obtain a more precise and simplified complexity analysis, as well as improved algorithm understanding.

1 Introduction

Model checking is a widely used technique for verifying that a property holds for a system. Systems to be verified can be modeled accurately by pushdown systems (PDS). Properties can be modeled by linear temporal logic (LTL) formulas. LTL is a language commonly used to describe properties of systems [12, 13, 21] and is sufficiently powerful to express many practical properties. Examples include many dataflow analysis problems and various correctness and security problems for programs.

This paper focuses on LTL model checking of PDS, specifically on the global model checking problem [15]. The model checking problem is formulated in terms of evaluation of a Datalog program [5]. Datalog is a database query language based on the logic programming paradigm [11, 1]. The Büchi PDS, corresponding to the product of the PDS and the automaton representing the inverse of the property, is expressed in Datalog facts, and a reach graph — an abstract representation of the Büchi PDS, is formulated in rules. The method described in [18] generates specialized algorithms and data structures and complexity formulas for the rules. The generated algorithms and data structures are such that

* This work was supported in part by NSF under grants CCR-0306399 and CCR-0311512 and ONR under grants N00014-04-1-0722 and N00014-02-1-0363.

** Corresponding author.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 190–206, 2006.
© Springer-Verlag Berlin Heidelberg 2006

given a set of facts, they compute all facts that can be inferred. The generated implementation employs an incremental approach that considers one fact at a time and uses a combination of linked and indexed data structures for facts. The running time is optimal, in the sense that each combination of instantiations of hypotheses is considered once in `O(1)` time.

Our main contributions are:

- A novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [5] and a systematic method for deriving efficient algorithms and data structures from the rules[18].
- A precise and automatic time complexity analysis of the model checking problem. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

We thus develop a model checker with improved time complexity guarantees and improved algorithm understanding.

The rest of this paper is organized as follows. Section 2 defines LTL model checking of PDS. Section 3 expresses the model checking problem by use of Datalog rules. Section 4 describes the generation of a specialized algorithm and data structures from the rules and analyzes time complexity of the generated implementation. Section 5 discusses related work and concludes.

2 Linear Temporal Logic Model Checking of Pushdown Systems

This section defines the problem of model checking PDS against properties expressed using LTL formulas, as described in [15].

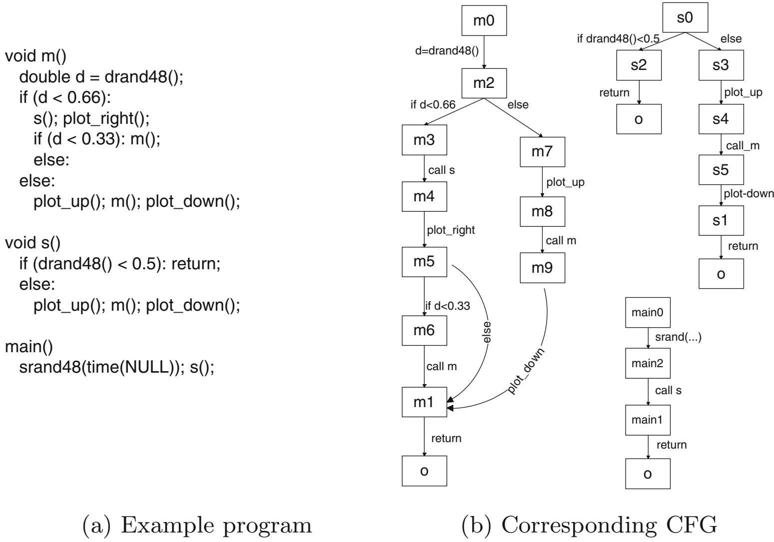
2.1 Pushdown Systems

A *pushdown system (PDS)* [14] is a triple (C_P, S_P, T_P), where C_P is a set of control locations, S_P is a set of stack symbols and T_P is a set of transitions. A transition is of the form $(c, s) \rightarrow (c', w)$ where c and c' are control locations, s is a stack symbol, and w is a sequence of stack symbols; it denotes that if the PDS is in control location c and symbol s is on top of the stack, the control location changes to c', s is popped from the stack, and the symbols in w are pushed on the stack, one at a time, from left to right. A *configuration* of a PDS is a pair (c, w) where c is a control location and w is a sequence of symbols from the top of the stack. If $(c, s) \rightarrow (c', w) \in T_P$ then for all $v \in {S_P}^*$, configuration (c, sv) is said to be an *immediate predecessor* of (c', wv). A *run* of a PDS is a sequence of configurations $conf_0, conf_1, ..., conf_n$ such that $conf_i$ is an immediate predecessor of $conf_{i+1}$, for $i = 0, \ldots, n-1$.

We only consider PDSs where each transition $(c, s) \rightarrow (c', w)$ satisfies $|w| \leq 2$. Any given PDS can be transformed to such a PDS. Any transition $(c, s) \rightarrow$

(c', w), such that $|w| > 2$, can be rewritten into $(c, s) \rightarrow (c', w_{hd}\ s')$ and $(c', s') \rightarrow (c, w_{tl})$, where w_{hd} is the first symbol in w, w_{tl} is w without its first symbol, and s' is a fresh symbol. This step can be repeated until all transitions have $|w| \leq 2$. This replaces each transition $(c, s) \rightarrow (c', w)$, where $|w| > 2$, with $|w| - 1$ transitions and introduces $|w| - 1$ fresh stack symbols.

The procedure calls and returns in a program correspond to a PDS [16]. First, we construct a control flow graph (CFG) [2] of the program. Then, we set up one control location, say called c. Each CFG vertex is a stack symbol. Each CFG edge (s, s') corresponds to a transition (i) $(c, s) \rightarrow (c, \epsilon)$, where ϵ stands for the empty string, if (s, s') is labeled with a return statement; (ii) $(c, s) \rightarrow (c, s' f_0)$, if (s, s') is labeled with a call to procedure f, and f_0 is f's entry point; (iii) $(c, s) \rightarrow (c, s')$, otherwise. A run of the program corresponds to a PDS run.



(a) Example program (b) Corresponding CFG

Fig. 1. Example program and corresponding CFG

Figure 1 shows an example program and its CFG [15]. The program creates random bar graphs using the commands `plot_up`, `plot_right`, and `plot_down`. The corresponding PDS is:

$$
\begin{aligned}
C_P &= \{c\} \\
S_P &= \{m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, s0, s1, s2, s3, s4, s5, \\
&\quad\ main0, main1, main2\} \\
T_P &= \{(c, m3) \rightarrow (c, m4s0), (c, m6) \rightarrow (c, m1m0), (c, m8) \rightarrow (c, m9m0), \\
&\quad (c, m1) \rightarrow (c, \epsilon), (c, s2) \rightarrow (c, \epsilon), (c, s4) \rightarrow (c, s5m0), \\
&\quad (c, s1) \rightarrow (c, \epsilon), (c, main2) \rightarrow (c, main1s0), (c, main1) \rightarrow (c, \epsilon)\}
\end{aligned}
$$

2.2 Linear Temporal Logic Formulas

Linear temporal logic (LTL) formulas [12, 13, 21] are evaluated over infinite sequences of symbols. The standard logic operators are available; if f and g are formulas, then so are $\neg f$, $f \wedge g$, $f \vee g$, $f \rightarrow g$. The following additional operators are available: $X\ f$: f is true in the next state; $F\ f$: f is true in some future state; $G\ f$: f is true globally, i.e. in all future states; $g\ U\ f$: g is true in all future states until f is true in some future state.

A LTL formula can be translated to a Büchi automaton, a finite state automaton over infinite words. The automaton accepts a word if on reading it a good state is entered infinitely many times. Formally, a *Büchi automaton* (BA) is a tuple $(C_B, L_B, T_B, C0_B, G_B)$ where C_B is a set of states, L_B is a set of transition labels, T_B is a set of transitions, $C0_B \subseteq C_B$ is a set of starting states, and $G_B \subseteq C_B$ is a set of good states. A transition is of the form (c, l, c'), where $c, c' \in C_B$ and $l \in L_B$. The label of a transition is a condition that must be met by the current symbol in the word being read, in order for the transition to be possible. A label _ denotes an unconditional transition. An *accepting run* of a Büchi automaton is an infinite sequence of transitions $(c_0, l_0, c_1), (c_1, l_1, c_2), \ldots, (c_{n-1}, l_{n-1}, c_n)$, where a state $c_i \in G_B$ appears infinitely many times.

To specify a program property using an LTL formula, the program's CFG edges are used as atomic propositions. LTL formulas are defined with respect to infinite runs of the program. The corresponding BA accepts an infinite sequence of CFG edges, if on reading it, the automaton enters a good state infinitely many times. For example, the property that plotting up is never immediately followed by plotting down is expressed by the LTL formula $F = G(plot_up \rightarrow X(\neg plot_down))$. The BA[1] corresponding to $\neg F$ is shown in Figure 2. In the diagram nodes correspond to states and edges correspond to transitions of the BA; double circles mark good states and a square marks the start state.

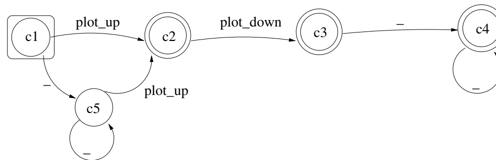


Fig. 2. Büchi automaton corresponding to $\neg G(plot_up \rightarrow X(\neg plot_down))$

2.3 LTL Model Checking of PDS

Given a system expressed as a PDS P, and a LTL formula F, the formula F holds for P if it holds for every run of P. We check whether F holds for P as follows [15]. First, we construct B — the BA corresponding to $\neg F$. Second, we

[1] The Büchi automaton was generated with the tool LBT that translates LTL formulas to Büchi automata (http://www.tcs.hut.fi/Software/maria/tools/lbt/).

construct BP — a Büchi PDS that is the product of P and B, and make sure BP has no accepting run. A *Büchi PDS* (BPDS) is a tuple (C, S, T, C_0, G), where C is a set of control locations, S is a set of stack symbols, T is a set of transitions, $C_0 \subseteq C$ is the set of starting control locations, $G \subseteq C$ is the set of good control locations. Transitions are of the form $((C * S) * (C * S^*))$. The concepts *configuration*, *predecessor*, and *run* of a BPDS are analogous to those of a PDS. An *accepting run* of the BPDS is an infinite sequence of configurations in which configurations with control locations in G appear infinitely many times. The product BPDS BP of $P = (C_P, S_P, T_P)$ and $B = (C_B, L_B, T_B, C0_B, G_B)$ is the five-tuple $((C_P * C_B), S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, where $(((c_P, c_B), s), ((c'_P, c'_B), w)) \in T_{BP}$ if $(c_P, s) \rightarrow (c'_P, w) \in T_P$, and there exists f such that $(c_B, f, c'_B) \in T_B$, and f is true at configuration $((c_P, c_B), s)$; $(c_P, c_B) \in C0_{BP}$ if $c_B \in C0_B$; $(c_P, c_B) \in G_{BP}$ if $c_B \in G_B$.

Next we construct a *reach graph* — a finite graph that abstracts BP. The nodes of the graph are configurations of BP. An edge $((c, s), (c', s'))$ in the reach graph corresponds to a run that takes BP from configuration (c, s) to configuration (c', s'). If a good control location in BP is visited in the run corresponding to an edge, the edge is said to be *good*. A path in the reach graph is a sequence of edges. Cycles in the reach graph correspond to infinite runs of BP. Paths containing cycles with good edges in them correspond to accepting runs of BP and are said to be *good*. If the reach graph corresponding to BP has no good paths, BP has no accepting runs and F holds for P. Otherwise, the good paths in the reach graph are counterexamples showing that F does not hold for P.

3 Specifying the Reach Graph in Rules and Detecting Good Paths

This section expresses the reach graph using Datalog rules and employs an algorithm for detecting good paths in the reach graph as presented in [5].

A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, ..., x_{1a_1}) \ \wedge ... \wedge \ p_h(x_{h1}, ..., x_{ha_h}) \rightarrow q(x_1, ..., x_a)$$

where h is a natural number, each p_i (respectively q) is a relation of a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $q(x_1, ..., x_a)$ is called a *fact*. The meaning of a set of rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules.

Expressing the Büchi PDS. The BPDS is expressed by the relations `loc`, `trans0`, `trans1`, and `trans2`. The `loc` relation represents the control locations of the BPDS; its arguments are a control location and a boolean argument indicating whether the control location is good. One instance of the relation

exists for each control location. The three relations `trans0`, `trans1`, and `trans2` express transitions. The facts `trans0(c1,s1,c2)`, `trans1(c1,s1,c2,s2)`, and `trans2(c1,s1,c2,s2,s3)`, where `ci`'s are control locations and `si`'s are stack symbols, denote transitions of the form of the form $((c, s), (c, w))$ such that, $w \in S^*_{BP}$ and $|w| = 0$, $|w| = 1$, and $|w| = 2$, respectively. `or` is a relation with three boolean arguments; in the fact `or(x1,x2,r)`, the argument `r` is the value of the logical *or* of the arguments `x1` and `x2`.

Expressing the edges of the reach graph. The reach graph is expressed by relations `erase` and `edge`. The fact `erase(c1,s1,g,c2)` denotes a run of BP from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$. The third element in the tuple is a boolean value that indicates whether the corresponding run goes through a good control location. The `edge` relation represents the reach graph edges. `edge(c1,s1,g,c2,s2)` denotes an edge between nodes $(c1, s1)$ and $(c2, s2)$; `g` is a boolean argument indicating whether the edge is good. For a BPDS $(C_{BP}, S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, `erase` and `edge` are the relation satisfying:

i. $(c1, s, g, c2) \in$`erase` if $(c1, s) \to (c2, \epsilon) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
ii. $(c1, s1, g1 \vee g2, c3) \in$`erase` if $(c1, s1) \to (c2, s2) \in T_{BP}$, and $(c2, s2, g2, c3) \in$ `erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise
iii. $(c1, s1, g1 \vee g2 \vee g3, c4) \in$`erase` if $(c1, s1) \to (c3, s2s3) \in T_{BP}$, $(c2, s2, g2, c3)$ $\in$`erase`, and $(c3, s3, g3, c4) \in$`erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise

and

i. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \to (c2, s2) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
ii. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \to (c2, s2s3) \in T_{BP}$, $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise
iii. $(c1, s1, g1 \vee g2, c3, s3) \in$`edge` if $(c1, s1) \to (c2, s2s3) \in T_{BP}$, $(c2, s2, g2, c3) \in$ `erase`, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

In model checking of programs, the relation `erase` summarizes the effects of procedures. The three parts of the above definition correspond to the program execution exiting, proceeding within, or entering a procedure.

The definitions of the `erase` and `edge` relations can be readily written as rules. These rules are shown in Figure 3.

Detecting good paths. Checking that the BPDS accepts the empty language amounts to checking that the resulting reach graph has no good paths. To find good paths in the reach graph we use the algorithm presented in [5,-Figure 4]but ignore consideration of resource labels by the algorithm. The algorithm uses depth first search and is linear in the number of edges in the reach graph.

```
trans0(c1,s1,c2)∧loc(c1,g)→erase(c1,s1,g,c2)
trans1(c1,s1,c2,s2)∧erase(c2,s2,g2,c3)∧loc(c1,g1)∧or(g1,g2,g)
    →erase(c1,s1,g,c3)
trans2(c1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)∧erase(c3,s3,g3,c4)∧
    loc(c1,g1)∧or(g1,g2,g4)∧or(g4,g3,g)→erase(c1,s1,g,c4)
trans1(c1,s1,c2,s2)∧loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3)∧loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)∧loc(c1,g1)∧or(g1,g2,g)
    →edge(c1,s1,g,c3,s3)
```

Fig. 3. Rules corresponding to the `erase` relation used to construct the reach graph, and the `edge` relation of the reach graph

4 Efficient Algorithm for Computing the Reach Graph

This section describes the generation of a specialized algorithm and datastructures for computing the reach graph from the rules shown in the previous section, as well as analyzing precisely the time complexity for computing the reach graph and expressing the complexity in terms of characterizations of the facts—the parameters characterizing the BPDS.

4.1 Generation of Efficient Algorithms and Data Structures

Transforming the set of rules into an efficient implementation uses the method in [18]. We first transform each rule with more than two hypotheses into multiple rules with two hypotheses each and then carry out three key steps. Step 1 transforms the least fixed point (LFP) specification of the rule set to a `while`-loop. Step 2 transforms expensive set operations in the loop into incremental operations. Step 3 designs appropriate data structures for each set, so that operations on it can be implemented efficiently. These three steps correspond to dominated convergence [10], finite differencing [20], and real-time simulation [19], respectively, as studied by Paige et al.

Auxiliary relations. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. The transformation introduces auxiliary relations with necessary arguments to combine two hypotheses at a time. We repeatedly apply the following transformations to each rule with more than two hypotheses until only rules with at most two hypotheses are left. We replace any two hypotheses of the rule, say $P_i(X_{i1},...,X_{ia_i})$ and $P_j(X_{j1},...,X_{ja_j})$ by a new hypothesis, $Q(X_1,...,X_a)$, where Q is a fresh relation, and X_k's are variables in the arguments of P_i or P_j that occur also in the arguments of other hypotheses or the conclusion of this rule. We add a new rule:

$$P_i(X_{i1},...,X_{ia_i}) \wedge P_j(X_{j1},...,X_{ja_j}) \rightarrow Q(X_1,...,X_a).$$

The resulting rule set for constructing the reach graph is shown in Figure 4. Several auxiliary relations have been introduced. The relations `gtrans1` and

`gtrans2` represent transitions like `trans1` and `trans2` respectively, but an extra argument indicates whether the transitions start at a good control location. The relations `gtrans1e` and `gtrans2e`, represent runs of the BPDS starting with a transition `trans1` and `trans2` respectively, followed by a run represented as a fact of the `erase` relation. The facts `gtrans1e(c1,s1,c2,g1,g2)` and `gtrans2e(c1,s1,s2,c2,g1,g2)` represent runs from configuration $(c1, s1)$ to configurations $(c2, \epsilon)$ and $(c2, s2)$ respectively, where `g1` and `g2` indicate, respectively, whether the first control location in the run is good and whether the rest of the run visits a good control location. The relation `gtrans2ee` represents runs consisting of one transition and two runs expressed as facts of the `erase` relation. The fact `gtrans2ee(c1,s1,c2,g1,g2,g3)` stands for a run from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$; the arguments `g1`, `g2`, and `g3` are booleans indicating respectively, whether the first control location in the run is good, and whether the remaining two parts of the run visit a good control location. The relations `gtrans1ee_or` and `gtrans2ee_or` represents runs like `gtrans1ee` and `gtrans2ee`, except with two boolean arguments combined using logical or.

```
1.  loc(c1,g)∧trans0(c1,s1,c2)→erase(c1,s1,g,c2)
2.  loc(c1,g1)∧trans1(c1,s1,c2,s2)→gtrans1(c1,g1,s1,c2,s2)
3.  gtrans1(c1,g1,s1,c2,s2)∧erase(c2,s2,g2,c3)→gtrans1e(c1,s1,c3,g1,g2)
4.  gtrans1e(c1,s1,c3,g1,g2)∧or(g1,g2,g)→erase(c1,s1,g,c3)
5.  loc(c1,g1)∧trans2(c1,s1,c2,s2,s3)→gtrans2(c1,g1,s1,c2,s2,s3)
6.  gtrans2(c1,g1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)
        →gtrans2e(c1,s1,s2,c3,g1,g2)
7.  gtrans2e(c1,s1,s2,c3,g1,g2)∧erase(c3,s2,g3,c4)
        →gtrans2ee(c1,s1,c4,g1,g2,g3)
8.  gtrans2ee(c1,s1,c4,g1,g2,g3)∧or(g1,g2,g4)
        →gtrans2ee_or(c1,s1,c4,g3,g4)
9.  gtrans2ee_or(c1,s1,c4,g3,g4)∧or(g4,g3,g)→ erase(c1,s1,g,c4)
10. gtrans1(c1,g,s1,c2,s2)→ edge(c1,s1,g,c2,s2)
11. gtrans2(c1,g,s1,c2,s2,s3)→ edge(c1,s1,g,c2,s2)
12. gtrans2e(c1,s1,s2,c2,g1,g2)∧or(g1,g2,g)→edge(c1,s1,g,c2,s2)
```

Fig. 4. The reach graph expressed in rules with at most two hypotheses

Fixed-point specification and `while`-loop. We represent a relation the form `Q(a1, a2, ... , an)` using tuples of the form `[Q,a1,a2,...,an]`. We use S `with` X and S `less` X to mean $S \cup \{X\}$ and $S - \{X\}$, respectively. We use the notation $\{X : Y_1$ `in` $S_1, \ldots, Y_n$ `in` $S_n | Z\}$ for set comprehension. Each Y_i enumerates elements of S_i; for each combination of $Y_1, \ldots, Y_n$ if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If Z is omitted, it is implicitly the constant *true*.

`LFP`(S_0, F) denotes the minimum element S, with respect to the subset ordering $\subseteq$, that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$. We use standard control constructs `while`, `for`, `if`, and `case`, and we use indentation to indicate scope. We abbreviate $X := X$ `op`Y as X `op`$:= Y$.

We use set `bpds` for the set of facts representing the BPDS.

```
rbpds = {[loc,c1,g] : loc(c1,g) in bpds} ∪
  {[trans0,c1,s1,c2] : trans0(c1,s1,c2) in bpds} ∪
  {[trans1,c1,s1,c2,s2] : trans1(c1,s1,c2,s2) in bpds} ∪
  {[trans2,c1,s1,c2,s2,s3] : trans0(c1,s1,c2,s2,s3) in bpds },
```

Given any set of facts `R`, and a rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)`, referred to as *result set*, be the set of all facts that can be inferred by rule `n` given the facts in `R`. For example,

```
2gtrans1 = {[gtrans c1 s1 g c2 s2] : [loc c1 g] in R and
                                     [trans1 c1 g s1 c2 s2] in R},
10edge = {[edge c1 s1 g c2 s2] : [gtrans1 c1 g s1 c2 s2] in R}.
```

The meaning of the give facts and the rules used to compute the reach graph is:

```
LFP({},F), where F(R) = rbpds ∪ 1erase(R) ∪ 2gtrans1(R) ∪
        3gtrans1e(R) ∪ 4erase(R) ∪ 5gtrans2(R) ∪ 6gtrans2e(R) ∪
        7gtrans2ee(R) ∪ 8gtrans2ee_or(R) ∪ 9erase(R) ∪
        10edge(R) ∪ 11edge(R) ∪ 12edge(R).
```

This least-fixed point specification of computing the reach graph is transformed into the following `while`-loop:

```
R := {}; while exists x in F(R) - R
   R with := x;                                  (1)
```

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop.

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression *exp* in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in *exp* are initialized and updated, respectively.

The expensive expressions in constructing the reach graph are all result sets, such as `2grtrans1(R)`, and `F(R)-R`. We use fresh variables to hold each of their respective values and maintain the following invariants:

```
Ibpds = rbpds, I1erase = 1erase(R),
I2gtrans1 = 2gtrans1(R), I3gtrans1e = 3gtrans1e(R),
I4erase = 4erase(R), I5gtrans2 = 5gtrans2(R),
I6gtrans2e = 6gtrans2e(R), I7gtrans2ee = 7gtrans2ee(R),
I8gtrans2ee_or = 8gtrans2ee_or(R), I9erase = 9erase(R),
I10edge = 10edge(R), I11edge = 11edge(R), I12edge = 12edge(R),
W = F(R) - R.
```

`W` serves as the workset. As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I2gtrans1`.

`I2gtrans1` is the value of the set formed by joining elements from the set of facts of the `loc` and `trans1` relations. `I2gtrans1` can be initialized to {} with the initialization `R = {}`. To update `Igtrans1` incrementally with update `R with:= x`, if `x` is of the form `[loc,c1,g]` we consider all matching tuples of the form `[trans1,c1,s1,c2,s2]` and add the tuple `[gtrans1,c1,g,s1,c2,s2]` to `I2gtrans1`. To form the tuples to add, we need to efficiently find the appropriate values of variables that occur in `[trans1,c1,s1,c1,s2]` tuples, but not in `[loc,c1,g]`, i.e. the values of `s1,c2,` and `s2`, so we maintain an auxiliary map that maps `[c1]` to `[s1,c2,s2]` in the variable `I2gtrans1_trans1` shown below. Symmetrically, if `x` is a tuple of `[trans1,c1,s1,c2,s2]`, we need to consider every matching tuple of `[loc,c1,g]` and add the corresponding tuple of `[gtrans1,c1,g,s1,c2,s2]` to `I2gtrans1_loc`. The first set of elements in auxiliar maps is referred to as the *anchor* and the second set of elements as the *nonanchor*.

```
I2gtrans1_trans1 = {[[c1], [s1,c2,s2]] :
                              [trans1,c1,s1,c2,s2] in R},
I2gtrans1_loc = {[[c1], [g]] : [loc,c1,g] in R}.
```

Thus, we are able to directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, as well as consider each combination only once. Similarly, such auxiliary maps are maintained for all invariants that we maintain.

All variables holding the values of expensive computations listed above and auxiliary maps are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. When `R` is {}, `Ibpds = rbpds`, all auxiliary maps are initialized to {}, and `W = Ibpds`. When a fact is added to `R` in the loop body, the variables are updated. We show the update for the addition of a fact of relation `trans1` only for `I2gtrans1` invariant and `I2gtrans1_loc` auxiliary map , since other facts and updates to the variables and auxiliary maps are processed in the same way. The notation `E{Ys}`, where `E = {[Ys,Xs]}` is an auxiliary map, is used to access all matching tuples of `E` and return all matching values of `Xs`.

```
case of x of [loc,c1,g]:
I2gtrans1 +:= {[gtrans1,c1,g,s1,c2,s2]:
    [s1,c2,s2] in I2gtrans1_trans1{c1}};
W +:= {[gtrans1,c1,g,s1,c2,s2]: [s1,c2,s2] in I2gtrans1_trans1{c1}
    |[gtrans1,c1,g,s1,c2,s2] notin R};
I2gtrans1_loc with:= {[[c1], [g]] : [loc,c1,g] in R};
```
(2)

Using the above initializations and updates, and replacing all invariant maintenance expressions with `W`, we obtain the following complete code:

```
initialization; R:={};
while exits x in W:
   update; W less:= x; R with:= x;
```
(3)

We next eliminate dead code and clean up the code to contain only uniform operations and set elements for data structure design. We then decompose `R` and `W` into several sets, each corresponding to a single relation that occurs in the rules. `R` is decomposed to `Rtrans0`, `Rtrans1`, `Rtrans2`, `Rloc`, `Rerase`, `Rgtrans1`, `Rgtrans1e`, `Rgtrans2`, `Rgtrans2e`, `Rgtrans2ee`, `Rgtrans2ee_or`, and `Redge`. `W` is decomposed in the same way. We eliminate relation names from the first component of tuples and transform the `while`-clause and `case`-clause appropriately. Then, we do the following three sets of transformations. We transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for`-loop that adds the elements one at a time. For example, `I2gtrans1 +:= {[gtrans1,c1,g,s1,c2,s2]: [s1,c2,s2] in I2gtras1_trans1{c1}}` is transformed into:

```
for [s1,c2,s2] in I2gtras1_trans1{c1}:
    I2gtrans1 +:= [c1,g,s1,c2,s2];
```

We replace tuples and tuple operations with maps and map operations. We make all element addition and deletion easy by testing membership first.

Data structures. After the above transformations each firing of a rule takes a constant number of set operations. Since each of these set operations takes worst case constant time in the generated code, achieved as described below, each firing of a rule takes worst case constant time. Next we describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M(X)$, element retrieval `for` X `in` S and `while exists` X `in` S, membership test X `in` S, X `notin` S, and element addition S `with` X and deletion S `less` X. We use *associative access* to refer to membership test and computing image set.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that are traversed by loops and both arrays and linked lists when both operations are needed.

The result sets, such as `Rtrans0`, are represented by nested array structures. Each of the result sets of, say, `a` components is represented using an `a`-level nested array structure. The first level is an array indexed by values in the domain of the first component of the result set; the k-th element of the array is null if there is no tuple of the result set whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an `(a-1)`-level nested array structure for remaining components of tuples of result sets whose first component has value `k`.

The worksets, such as `Wtrans0`, are represented by arrays and linked lists. Each workset is represented the same as the corresponding resultset with two additions. First, for each array we add a linked list linking indices of non-null elements of the array. Second, to each linked list we add a tail pointer. One or more records are used to put each array, linked list, and tail pointer together. Each workset is represented simply as a nested queue structure (without the underlying arrays), one level for each workset, linking the elements (which correspond to indices of the arrays) directly.

Auxiliary maps, such as `I2gtrans1_trans1` and `I2gtrans1_loc`, are implemented as follows. Each auxiliary map, say `E` for a relation that appears in a rule's conclusion uses a nested array structure as resultsets and worksets do and additionally linked lists for each component of the non-anchor as worksets do. `E` uses a nested array structure only for the anchor, where elements of the arrays of the last component of the anchor are each a nested linked-list structure for the non-anchor.

4.2 Complexity Analysis of the Model Checking Problem

We analyze the time complexity of the model checking problem by carefully bounding the number of facts actually used by the rules. For each rule we determine precisely the number of facts processed by it, avoiding approximations that use the sizes of individual argument domains.

Calculating time complexity. We first define the size parameters used to characterize relations and analyze complexity. For a realtion `r` we refer to the number of facts of `r` that are given or can be inferred as `r`'s *size*. The parameters `#trans0`, `#trans1` and `#trans2` denote the number of transitions of the form $((c1, s1), (c2, \epsilon))$, $((c1, s1), (c2, s2))$, and $((c1, s1), (c2, s2s3))$, respectively; `#trans` denotes the total number of transitions. The parameters `#gtrans1` and `#gtrans2` denote the number of facts of relations `gtrans1` and `gtrans2`, where `#gtrans1`=`#trans1` and `#gtrans2`=`#trans2`. Parameters `#gtrans1e` and `#gtrans2e` denote the relation sizes — `#trans1` $*$ `#target_loc_trans0`, and `#trans2` $*$ `#target_loc_trans0`, respectively, and `#gtrans2ee` denotes the corresponding relation size equal to `#trans2` $*$ `#target_loc_trans0`^{2}. The parameter `#erase` denotes the number of facts in the `erase` relation; `#erase.4/123` denotes the number of different values the forth argument of `erase` can take for each combination of values of the first three arguments. In the worst case, this is the number of control locations $c2$ such that a transition of the form $((c1, s1), (c2, \epsilon))$ exists in the automaton. We use `#target_loc_trans0` to denote this number.

The time complexity for the set of rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule `r`, `r.#firedTimes` stands for the number the number of firings for the rule is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts which make the two hypotheses simultaneously true. The total time complexity is time for reading the input, i.e. `O(#trans + #loc)`, plus the time for applying each rule, shown in the second column in the table of Figure 5.

Time complexity of model checking PDS. Time complexity for processing each of the rules and computing the `erase` and `edge` relations is shown in the second table of Figure 5. After the reach graph has been computed, good cycles in the reach graph can be detected in time linear in the size of the reach graph, i.e. `O(#edge)`. Thus, the asymptotic complexity of the model checking problem is dominated by the time complexity of computing the `erase` relation.

rule no	time complexity	time complexity bound
1	min(#trans0*1,#loc*#trans0.23/1)	#trans0
2	min(#loc*#trans1.234/1,#trans1*1)	#trans1
3	min(#gtrans1*#erase.4/123, #erase*#gtrans1.12/34)	#trans1*#target_loc_trans0
4	min(#gtrans1e*1, 1*#gtrans1e)	#trans1*#target_loc_trans0
5	min(#loc*#trans2.2345/1,#trans2*1)	#trans2
6	min(#gtrans2*#erase.4/123, #erase*#gtrans2.12/345)	#trans2*#target_loc_trans0
7	min(#gtrans2e*#erase.4/123, #erase*#gtrans2e.12/345)	#trans2*#target_loc_trans0^2
8	min(#gtrans2ee*1,1*#gtrans2ee)	#trans2*#target_loc_trans0^2
9	min(#gtrans2ee_or*1,1*#gtrans2ee_or)	#trans2*#target_loc_trans0^2
10	min(#gtrans2ee_or*1,1*#gtrans2ee_or)	#trans2*#target_loc_trans0^2
11	#gtrans1	#trans1
12	#gtrans2	#trans2
13	min(#gtrans2e*1,1*#gtrans2e)	#trans2*#target_loc_trans0

relation	time complexity
erase	O(#trans0 + #trans1*#target_loc_trans0 + #trans2*#target_loc_trans0^2)
edge	O(#trans1 + #trans2*#target_loc_trans0)

Fig. 5. Time complexity of computing the reach graph

For a BPDS, product of $P = \{C_P, S_P, T_P\}$ where $|C_P| = 1$, and $B = \{C_B, L_B, T_B, C0_B, G_B\}$, #target_loc_trans0$\leq|C_B|$, and #trans2$\leq|T_P| * |T_B|$. For such a PDS, $O(|T_P| * |T_B| * |C_B|^2)$ is the worst case time complexity of computing the erase relation and $O(|T_P| * |T_B| * |C_B|)$ is the worst case time complexity for computing the edge relation. Since only $|T_P|$ is dependent on the size of P, time complexity is linear in the size of the P and cubic in the size of B.

4.3 Performance

We tested the performance of our reach graph construction algorithm on two sets of BPDS consisting of BPDS with increasing #trans. BPDS in one set also had increasing #target_loc_trans0, while BPDS in the second set had constant #target_loc_trans0. The time complexity for computing reach graphs for BPDS in the first set is as shown in Figure 5. However, for automata in the second set time complexity should be linear — O(#trans). If the PDS corresponds to a program, #target_loc_trans0 is proportional to the total number of return points of procedures in the program. Thus, our test data corresponds to checking if a property holds on programs with an increasing number of statements and procedure calls, and programs with an number of statements, but constant number of procedures.

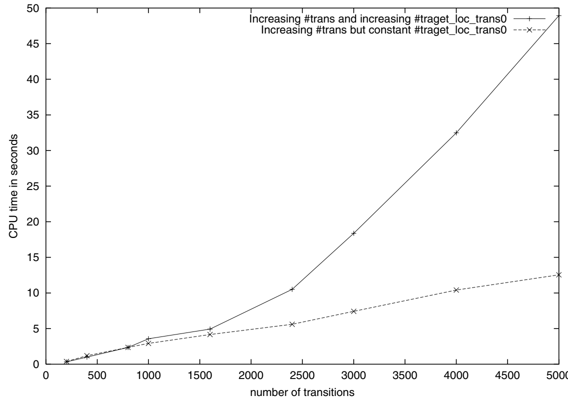


Fig. 6. Results for computing the reach graph for the BPDS

Results of the experiment are shown in Figure 6 and confirm our analysis. We used generated python code in which each operation on set elements is guaranteed to be constant time on average using default hashing in python. Running times are measured in seconds on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running SunOS 5.8. Running times are the average over ten runs.

5 Discussion

The problem of LTL model checking of PDS has been extensively researched, especially model checking PDS induced by CFGs of programs. The model checking problem for context-free and pushdown processes is explored in [8]. The design and implementation of Bebop: a symbolic model checker for boolean programs, is presented in [4]. Burkart and Steffen [9] present a model checking algorithm for modal mu-calculus formulas. For a PDS with one control state, a modal-mu calculus formula of alternation depth k can be checked in time $O(n^k)$, where n is the size of the PDS. The works [17, 16, 15, 7] describe efficient algorithms for model checking PDSs. Alur et al. [3] and Benedikt et al. [6] show that state machines can be used to model control flow of sequential programs. Both works describe algorithms for model checking PDS that have time complexity cubic in size of the BA and linear in size of the PDS; these works combine forward and backward reachability and obtain complexity estimations by exploiting this mixture. Esparza et al. [15] estimate time complexity of solving the model checking problem to be `O(n*m`^{3}`)` for model checking PDS with one state only, where `n` is the size of the PDS and `m` is the size of the property BA [15]. While this is also linear in the size of the PDS, our time complexity analysis is more precise and automatic.

The algorithm derived in this work is essentially the same as the one in [15]. What distinguishes our work is that we use a novel implementation strategy

for the model checking problem that combines an intuitive definition of the model checking problem in rules [5] and a systematic method for deriving efficient algorithms and data structures from the rules [18], and arrives at an improved complexity analysis. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

An implementation of the model checking problem in logical rules is presented in [5]. The rules are evaluated using the XSB system [23]. Thus, the efficiency of the computation is highly dependent on the order of hypotheses in the given rules. Our implementation is drastically different, as it finds the best order of hypotheses in the rules automatically. We do not employ an evaluation strategy for Datalog, but generate a specialized algorithm and implementation directly from the rules.

In this paper, we presented an efficient algorithm for LTL model checking of PDS. We showed the effectiveness of our approach by using a precise time complexity analysis, along with experiments. These results show that our model checking algorithm can help accommodate larger PDS and properties. Our work is potentially a contribution not only to the model checking problem, since the idea behind the `erase` relation and the reach graph is more universal than model checking PDS. Variants of the `erase` relation are used in data flow analysis techniques, as described in [22] and related work. Applications of model checking in dataflow analysis are presented in [25, 24]. It is a topic of future research to apply our method to dataflow analysis problems.

Acknowledgment. Thanks to Tom Rothamel for helping debug performance problems in the implementation.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
3. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 207–220, London, UK, 2001. Springer-Verlag.
4. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
5. S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, London, UK, 2002. Springer-Verlag.
6. M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 652–666, London, UK, 2001. Springer-Verlag.

7. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.
8. O. Burkart, D. Caucal, F. Moller, and B. Steffen. *Verification on infinite structures.* North Holland, 2000.
9. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 419–429, London, UK, 1997. Springer-Verlag.
10. J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.
11. S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.
12. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.
14. J. Edmund M. Clark, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 1999.
15. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 232–247, London, UK, 2000. Springer-Verlag.
16. J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 324–336, London, UK, 2001. Springer-Verlag.
17. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, volume 9 of *Electronic Notes in Theoretic Comp. Sci.* Elsevier, 1997.
18. Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 172–183. ACM Press, 2003.
19. R. Paige. Real-time simulation of a set machine on a ram, 1989.
20. R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Program. Lang. Syst.*, 4(3):402–454, 1982.
21. J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

23. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data SIGMOD'94*, pages 442–453, 1994.
24. B. Steffen. Generating data flow analysis algorithms from modal specifications. In *TACS'91: Selected papers of the conference on Theoretical aspects of computer software*, pages 115–139, Amsterdam, The Netherlands, 1993. Elsevier Science Publishers B. V.
25. B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *CONCUR '95: Proceedings of the 6th International Conference on Concurrency Theory*, pages 72–87, London, UK, 1995. Springer-Verlag.

A Logic and Decision Procedure for Predicate Abstraction of Heap-Manipulating Programs*

Jesse Bingham and Zvonimir Rakamarić

Department of Computer Science, University of British Columbia, Canada
jesse.d.bingham@intel.com
zrakamar@cs.ubc.ca

Abstract. An important and ubiquitous class of programs are *heap-manipulating programs* (HMP), which manipulate unbounded linked data structures by following pointers and updating links. *Predicate abstraction* has proved to be an invaluable technique in the field of software model checking; this technique relies on an efficient decision procedure for the underlying logic. The expression and proof of many interesting HMP safety properties require *transitive closure* predicates; such predicates express that some node can be reached from another node by following a sequence of (zero or more) links in the data structure. Unfortunately, adding support for transitive closure often yields undecidability, so one must be careful in defining such a logic. Our primary contributions are the definition of a simple transitive closure logic for use in predicate abstraction of HMPs, and a decision procedure for this logic. Through several experimental examples, we demonstrate that our logic is expressive enough to prove interesting properties with predicate abstraction, and that our decision procedure provides us with both a time and space advantage over previous approaches.

1 Introduction

In recent years *software model checking* has emerged as a vibrant area of formal verification research. Much of the success of applying model checking to software has come from the use of *predicate abstraction* on the program source [16, 14, 3, 18]. In predicate abstraction, sets of states of the program and program transitions are over-approximated using a finite set of predicates over the program variables. These predicates (or boolean combinations thereof) typically express features of the program under verification such as its conditionals and relevant propositions about its variables. An integral ingredient in predicate abstraction is a decision procedure for the logic of the predicates. Since most approaches involve many queries to this decision procedure, performance is paramount.

An important class of programs are those we call *heap-manipulating programs* (HMPs), which are programs that access and modify linked data structures consisting of an unbounded number of uniform *heap nodes*. HMPs access the heap nodes through a finite number of pointers (that we call *node variables*) and following pointer fields between nodes. To apply predicate abstraction to HMPs and assert many interesting correctness properties, one must be able to express the concept of unbounded *reachability* (a.k.a. *transitive closure*) between nodes. This is done through a binary operator

* The authors were supported by grants from the Natural Sciences and Research Council of Canada (NSERC). Jesse Bingham has moved to Intel Corporation, Hillsboro, Oregon, U.S.A.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 207–221, 2006.
© Springer-Verlag Berlin Heidelberg 2006

that takes two node terms x and y, and asserts that the second can be reached from the first by following zero or more links; in our syntax this is written as $f^*(x,y)$ (f is the name of the *link function*). For example, $f^*(f(x),x)$ expresses that x is a node in a circular linked list.

Several papers have previously identified the importance of transitive closure for HMPs [30, 31, 5, 19, 2, 23]. Unfortunately, adding support for transitive closure to even relatively tame logics often yields undecidability [19]. Our first contribution is a fragment of the decidable logics that we show (through several nontrivial experiments) is still expressive enough to verify properties of interest for HMPs using predicate abstraction. Decidability of our logic follows from a small model theorem, akin to that of Benedikt et al. [5] and Balaban et al. [2], which states that if a set of predicates is satisfiable, then it is satisfiable by a heap structure with some bounded number of nodes. A naive decision procedure can thus enumerate all the (super-factorial but finite) number of structures of size up to this bound. We do not formally state or prove a small model theorem in this paper, rather, our second and most important contribution is an efficient decision procedure for our logic. We show that this procedure, though a worst case exponential time algorithm, solves the vast majority of queries sent to it during predicate abstraction very quickly. The result is an approach that can have large time and memory savings over decision procedures that enumerate all models, even when BDDs are used for this enumeration, as done by Balaban et al. [2].

The paper is organized as follows. Sect. 2 summarizes other work on verification of HMPs. Predicate abstraction and our verification framework (based on previous work), is outlined in Sect. 3. HMPs are introduced in Sect. 4. Sects. 5 and 6 respectively define our transitive closure logic and the decision procedure. We present experimental results in Sect. 7. Sect. 8 draws conclusions and discusses several important extensions to our logic and decision procedure that we believe are possible, but have been left as future work. Please note that our technical report [6] provides proofs of the theorems, additional details regarding the decision procedure, pseudocode for the example programs, and the sets of predicates needed for their verification.

2 Related Work

Balaban et al. [2] present an approach for shape analysis based on predicate abstraction that is similar to ours. The logic they use for describing properties of heap structures has slightly richer expressiveness than the logic we define in this paper.[1] The major difference between the two approaches is the way a program abstraction is computed. To compute the abstraction, they employ a small model theorem, and build BDDs representing all models up to the small model size. This is a bottleneck in both computation time and memory, since these BDDs tend to blow-up. The technique of Kesten and Pnueli [21] for establishing termination employed by Balaban et al. is likely compatible with our work also.

McPeak and Necula [28] specify heap data structures using *local equality axioms*, which constrain only a bounded fragment of the heap around some node. This enables them to describe a variety of shapes and reason about scalar values without abstracting

[1] Whereas our logic is unquantified, they allow restricted universal quantification.

them, while still preserving decidability. However, they can only approximate reachability between nodes (though *un*reachability is precise). When pointer disequalities are added, their decision procedure becomes incomplete. We handle both reachability and disequalities, but we can't describe such a variety of shapes. In addition, we compute an inductive invariant of a program automatically (given an appropriate set of predicates), while they require a user to provide loop invariants, which can be a significant burden.

The *Pointer Assertion Logic Engine* (PALE) [29] specifies heap structures using graph types [22], which are tree-shaped data structures augmented with extra pointers. The authors show that many common heap structures can be defined that way, some of which we cannot express. PALE relies on a decision procedure with non-elementary complexity, so, there are programs that cannot be verified in practice. Furthermore, loop invariants must be provided by the user.

The *Three Valued Logic Analyzer* (TVLA) [32, 25] extends conventional abstract interpretation with a third "uncertain" logic value, and builds so-called *3-valued logical structures* that abstract the reachable states at each program point (a.k.a. *canonical abstraction*). The abstract semantics of program statements are defined by *abstract transformers*, which can be generated by TVLA or user-defined if necessary. We cannot handle all heap structures that TVLA can, however, the abstract invariant we compute is always the most precise w.r.t. the given set of predicates. TVLA does not make such a guarantee, although some work has been done to make TLVA more precise [33]. TVLA is also employed by Manevich et al. [27], who observe that the number of shared nodes in linked lists is bounded and present a novel definition of "uninterrupted list segments". This is used to define predicate and canonical abstractions of potentially circular singly linked lists, and enables them to verify some HMPs that we are not able to verify, though their properties tend to be simpler than ours (see Sect. 7).

Lahiri and Qadeer [23] define two new predicates to express reachability of heap nodes in linked lists. To prove properties of HMPs, they use first-order axioms over those predicates. The given set of axioms is incomplete, and they provide an induction principle that is used to derive additional axioms when necessary. Because of the purely first-order axiomatization, they are able to harness the power of available automated theorem provers; they use UCLID [8] as the underlying inference engine.

Dams and Namjoshi [11] propose an approach based on predicate abstraction and model checking. They abstract a program by iteratively calculating weakest preconditions of shape predicates, and are able to handle second-order shape properties such as reachability, cyclicity, and sharing. The algorithm doesn't use a decision procedure, and as a consequence, new predicates can be generated in every iteration. Hence, the algorithm often has to be manually provided with "approximation hints" to converge.

3 Verification Approach

3.1 Predicate Abstraction

Our approach to verifying heap programs is based on *predicate abstraction* [16], which is an instance of *abstract interpretation* [10]. In the framework of abstract interpretation, a *concrete system* (in our case an HMP) is verified by constructing a finite-state over-approximation of the concrete system called the *abstract system*. Let $\mathscr{C}$ (the *concrete*

states) be the set of states of the concrete system. Predicate abstraction employs a finite set of predicates $\phi_1, \ldots, \phi_k$ in some logic that are assertions about concrete states. Corresponding to the predicates respectively are the *abstract boolean variables* $b_1, \ldots, b_k$. The set of *abstract states* $\mathscr{A}$ will be the set of assignments to the abstract boolean variables. The *abstraction function* $\alpha : \mathscr{C} \rightarrow \mathscr{A}$ is defined such that $\alpha(c)(b_i) = \mathsf{true}$ if and only if $c \models \phi_i$. A set of concrete states C is then abstracted by

$$\alpha(C) = \{\alpha(c) \mid c \in C\}$$

Note that since $\mathscr{A}$ is finite, $\alpha(C)$ is always finite as well. In contrast, $\mathscr{C}$ is often infinite; in our case the infinitude of concrete states arises from the unboundedness of the heap in HMPs.

Let $R \subseteq \mathscr{C}$ be the set of concrete states that are reachable in the concrete system. We wish to verify that a property expressed as a state assertion ψ over the concrete states holds for all members of R, i.e. that the implication $R \rightarrow \psi$ holds. Predicate abstraction is used to solve this problem by computing a set $R^\alpha \subseteq \mathscr{A}$ such that $\alpha(R) \subseteq R^\alpha$. Verification succeeds if one can prove that $R^\alpha \rightarrow \psi$. A key difference in the various approaches to predicate abstraction is how R^α is computed [16, 14, 12, 15, 2, 11]. This typically involves numerous queries to a decision procedure for the underlying logic and there are tradeoffs between how accurately R^α approximates $\alpha(R)$ and the number and complexity of these queries.

R^α is usually computed as a fixpoint of some approximation of the *abstract post image operator* $\mathsf{post} : 2^{\mathscr{A}} \rightarrow 2^{\mathscr{A}}$, defined as follows. Given a set of abstract states A, let

$$\mathsf{post}(A) \;=\; \left\{\alpha(c') \mid \exists c, c' \in \mathscr{C}.(c, c') \in T \wedge \alpha(c) \in A\right\}$$

where T is the transition relation of the concrete system. $\mathsf{post}(A)$ is thus the set of abstract states representing concrete states that are concrete successors of those states represented by A.

Since predicate abstraction is an incomplete approach, if it fails to verify the property, this can either happen because the concrete systems actually violates the property, or because of the loss of information inherent in the abstraction. Finding the "right" set of predicates for verification to go through can be tricky business. Many works have addressed this issue of *predicate discovery* [13, 4, 18, 11], which falls under the more general umbrella of *abstraction refinement* [9]. As in recent papers on this topic [2, 23], in our current framework, predicates are added by manual inspection of counterexample behaviors; applying automatic predicate discovery techniques is an important area of future work.

3.2 Computing post

Our tool computes post precisely; the algorithm can be viewed as an improvement over the following naive algorithm. Since post distributes over disjunction,[2] computing $\mathsf{post}(A)$ is reducible to computing $\mathsf{post}(\rho)$ for each cube ρ in some disjunctive normal form decomposition of A. Here, *cube* means a partial boolean assignment to

[2] Meaning that $\mathsf{post}(A_1 \vee A_2) = \mathsf{post}(A_1) \vee \mathsf{post}(A_2)$.

the abstract variables, and represents all abstract states that agree on this subset of the abstract variables.[3] By using a BDD [7] to represent A, we can easily obtain such a decomposition. The naive algorithm cycles through all 2^k abstract states a, and checks if $a \in \mathsf{post}(\rho)$; $\mathsf{post}(\rho)$ is then the BDD representing the disjunction of all such a. Each check of $a \in \mathsf{post}(\rho)$ involves a call to the decision procedure to determine if the following formula is satisfiable:

$$\gamma(\rho) \wedge \mathsf{wp}(\gamma(a)) \tag{1}$$

where γ is the *concretization function*, and wp is the *weakest precondition* operator [17]. Intuitively, γ maps a cube to a logic formula that denotes the set of concrete states represented by the cube. Formally, for a cube μ let $P(\mu)$ (resp. $N(\mu)$) denote the set $\{i \mid \mu(b_i) = \mathsf{true}\}$ (resp. $\{i \mid \mu(b_i) = \mathsf{false}\}$). Then define

$$\gamma(\mu) \quad = \quad \bigwedge_{i \in P(\mu)} \phi_i \;\wedge \bigwedge_{i \in N(\mu)} \neg\phi_i$$

The weakest precondition operator wp is a syntactic transformation on logic formulas that depends on the program statement under consideration [17]. For example, for an assignment statement $x := e$, where x is a variable and e is some expression, $\mathsf{wp}(\pi)$ is constructed by syntactically replacing all occurrences of x with e in the formula π.[4] Our approach applies wp at the granularity of individual program statements when performing predicate abstraction.

Das et al.'s computation of post that we employ uses several straightforward optimizations over this naive algorithm [14]. First, if (1) contains a syntactic contradiction, meaning the existence of a predicate and its negation, then clearly the formula is not satisfiable. In such circumstances there is no need to call the decision procedure. When computing $\mathsf{post}(\rho)$, our implementation initially computes a BDD C representing the set of all a that won't yield such a contradiction. Second, rather than enumerating all $a \in C$, we do recursive case-splitting on the abstract variables, which allows for pruning of large portions of C. For example, let μ be the cube that assigns true to b_1 and leaves all other variables unconstrained. Then if $\gamma(\rho) \wedge \mathsf{wp}(\gamma(\mu))$ is unsatisfiable, then so too is $\gamma(\rho) \wedge \mathsf{wp}(\gamma(a))$ for *any* abstract state a that has b_1 equal to true. Hence, our algorithm would only explore those abstract states having b_1 false.

4 Heap-Manipulating Programs

In our framework, the *heap* consists of an unbounded number of *nodes*. HMPs allow for node variables (pointers), data fields for nodes, a link field f for nodes, and all other variables are modelled (or encoded as) booleans.

In lieu of a formal presentation of HMPs, we give an example called ND-INSERT in Fig. 1 that captures most of the interesting features. This program takes a node *head* and a node *item*, and inserts *item* into the linked list pointed to by *head* at a position

[3] A *partial boolean assignment* maps each variable b_i to an element of $\{\mathsf{true}, \mathsf{false}, \mathsf{undef}\}$.

[4] This only works under the assumption that x cannot be aliased.

```
1: procedure ND-INSERT(head, item)
2:     assume ¬f*(head, item) ∧ f*(head, nil) ∧ ¬head = nil ∧ f(item) = nil ∧ p = head
3:     while true do
4:         if ND ∨ f(p) = nil then
5:             f(item) := f(p);
6:             f(p) := item;
7:             break
8:         else
9:             p := f(p);
10:        end if
11:    end while
12:    assert f*(head, item) ∧ f*(head, nil)
13: end procedure
```

Fig. 1. A program that nondeterministically inserts a node *item* into the list pointed to by *head*. Here *ND* is a boolean value that is nondeterministically true or false.

selected nondeterministically. *head* is assumed to be non-nil and to point to an acyclic linked list that does not contain *item*. These assumptions are formalized by the **assume** statement on line 2 of the program. In the **assume** statement, and also in the **assert** statement, the subformulas of the form $f^*(x,y)$ express that node y is reachable from node x by following a sequence of f links of any length; we will formally define these predicates in Sect. 5. The fact that nil is reachable from *head* enforces the acyclicality assumption.[5]

The body of ND-INSERT is straightforward; a pointer p walks the list, and *item* is inserted at some point. The loop breaks once the insertion has occurred. The expression *ND* represents a nondeterministic boolean value. *item* is inserted when either $ND =$ true, or the end of the list is reached (detected by the disjunct $f(p) = \mathsf{nil}$ on line 4). The specification is expressed by the **assert** statement on line 12, and indicates that whenever line 12 is reached, *head* must point to an acyclic list that contains *item*.

The verification problem we wish to solve can be stated as follows: given an HMP, determine whether it is the case that all executions that satisfy all **assume** statements also satisfy all **assert** statements. Since the number of nodes in the heap is unbounded, HMPs are generally infinite state, thus one cannot directly apply finite-state model checking to this problem without using abstraction.

5 A Simple Transitive Closure Logic

Our logic assumes finite sets of *node* variables V, *boolean* variables B, *data function* variables D, and a single *link function* symbol f. The *term*, *atom*, and *literal* syntactic entities are given in Fig. 2. Literals of the form $x = y$, $\neg x = y$, $f^*(x,y)$, and $\neg f^*(x,y)$ (where x and y are terms) are called *equality*, *inequality*, *reachability*, and *unreachability* literals, respectively. Literals of the form $d(x)$ or $\neg d(x)$, where $d \in D$, are called *data* literals, while those of the form b or $\neg b$ are called simply *boolean variable* literals.

[5] In our logical framework, nil is modelled simply as a node having $f(\mathsf{nil}) = \mathsf{nil}$.

$$
\begin{array}{rcl}
v & \in & V \\
d & \in & D \\
b & \in & B \\
\mathit{term} & ::= & v \mid f(\mathit{term}) \\
\mathit{atom} & ::= & f^*(\mathit{term},\mathit{term}) \mid \mathit{term}=\mathit{term} \mid d(\mathit{term}) \mid b \\
\mathit{literal} & ::= & \mathit{atom} \mid \neg \mathit{atom}
\end{array}
$$

Fig. 2. The syntax of our simple transitive closure logic

The structures over which the semantics of our logic is defined are called *heap structures*. A heap structure $H = (N, \Theta)$ involves a finite set of *nodes* N and a function Θ that interprets each symbol σ in $V \cup B \cup D \cup \{f\}$ such that

$$
\begin{array}{ll}
\Theta(\sigma) \in N & \text{if } \sigma \in V \\
\Theta(\sigma) \in \{\mathsf{true}, \mathsf{false}\} & \text{if } \sigma \in B \\
\Theta(\sigma) \in N \to \{\mathsf{true}, \mathsf{false}\} & \text{if } \sigma \in D \\
\Theta(\sigma) \in N \to N & \text{if } \sigma = f
\end{array}
$$

Thus Θ interprets each node variable as a node, each boolean variable as a boolean value, each data function variable as a function that maps nodes to booleans, and the link function f is interpreted as a mapping from nodes to nodes. Heap structures naturally model a linked data structure of nodes, each node having a single pointer to another node and some finite number of boolean-valued fields. The *size* of H is defined to be $|N|$. The variables of V model program variables that point to nodes in the data structure, while the variables of B model program variables of boolean type. Clearly, program variables or node fields of any finite enumerated type can be encoded using the booleans accommodated by our logic.

We extend Θ to Θ^e, which interprets any term or atom in the obvious way, formally defined here. The interpretation of a term τ is defined inductively by:

$$
\Theta^e(\tau) = \begin{cases} \Theta(\tau) & \text{if } \tau \in V \\ \Theta(f)(\Theta^e(\tau')) & \text{if } \tau \text{ has the form } f(\tau') \text{ for some term } \tau' \end{cases}
$$

Θ^e interprets atoms as boolean values. An equality atom $\tau_1 = \tau_2$ is interpreted as true by Θ^e iff $\Theta^e(\tau_1) = \Theta^e(\tau_2)$. A data atom is interpreted by defining $\Theta^e(d(\tau)) = \Theta(d)(\Theta^e(\tau))$. A reachability atom $f^*(\tau_1, \tau_2)$ is interpreted as true iff there exists some $n \geq 0$ such that $\Theta(f)^n(\Theta^e(\tau_1)) = \Theta^e(\tau_2)$.[6] Finally, a literal that is not an atom is of the form $\neg\phi$ where ϕ is an atom, and we simply define $\Theta^e(\neg\phi) = \neg\Theta^e(\phi)$.

Sticking to the usual notation, given a heap structure $H = (N, \Theta)$ and a literal ϕ, we write $H \models \phi$ iff $\Theta^e(\phi) = \mathsf{true}$. For a set of literals Φ, we write $H \models \Phi$ iff $H \models \phi$ for all $\phi \in \Phi$.

6 Decision Procedure

The decision problem we aim to solve with our decision procedure is this: given a finite set of literals Φ, does there exist a heap structure H such that $H \models \Phi$? If there is such an

[6] Here, function exponentiation represents iterative application: for a function g and an element x in its domain, $g^0(x) = x$, and $g^n(x) = g(g^{n-1}(x))$ for all $n \geq 1$.

$$\frac{}{v=v}\text{IDENT} \qquad \frac{}{f^*(v,v)}\text{REFLEX} \qquad \frac{f(x)=y}{f^*(x,y)}\text{TRANS1}$$

$$\frac{f^*(x,y) \quad f^*(y,z)}{f^*(x,z)}\text{TRANS2} \qquad \frac{f(x)=y \quad f^*(x,z)}{x=z \qquad f^*(y,z)}\text{FUNC}$$

$$\frac{f(x_1)=x_2 \quad f(x_2)=x_3 \quad \cdots \quad f(x_k)=x_1 \quad f^*(x_1,y)}{y=x_1 \qquad y=x_2 \qquad \cdots \qquad y=x_k}\text{CYCLE}_k$$

$$\frac{f^*(x,y) \quad f^*(y,x) \quad f^*(x,z)}{x=y \qquad f^*(z,x)}\text{SCC} \qquad \frac{f^*(x,y) \quad f^*(x,z)}{f^*(y,z) \qquad f^*(z,y)}\text{TOTAL}$$

$$\frac{f(x)=z \quad f(y)=z \quad f^*(x,y) \quad f^*(y,x)}{x=y}\text{SHARE}$$

Fig. 3. The set of inference rules. Here x, y, and z range over (not necessarily distinct) terms. In the rules IDENT and REFLEX, v is restricted to be a variable that is already mentioned; this restriction prevents either of these rules from introducing new terms. CYCLE$_k$ actually defines a separate rule for each $k \geq 1$.

H, then we say that Φ is *satisfiable*, otherwise Φ is *unsatisfiable*. Clearly, any algorithm for this problem can be used to decide the satisfiability of a conjunction-of-literals (1) by simply taking Φ to be the set of its conjuncts.

Decidability of this problem follows from a small model theorem enjoyed by our logic, akin to other transitive closure logics [2, 5]. Our small model theorem states that Φ is satisfiable if and only if there exists H of size at most n such that $H \models \Phi$, where n is the number of distinct terms mentioned in Φ. Hence, a decision procedure can simply enumerate the finite set of such H, and for each one check if $H \models \Phi$. However, since the number of such heap structures is at least n^n, this approach is impractical. Employing BDDs [7] to represent the set of heap structures that satisfy Φ [2] is also memory-intensive; building a BDD for the literal $f^*(x,y)$ over just 8 nodes cannot be done in 2 GB of memory. This stems from the fact that such a BDD must represent the multitude of different paths that could exist between the nodes $\Theta^e(x)$ and $\Theta^e(y)$.

Our approach has relatively small memory requirements, and is based on a set of inference rules (IRs) with the property that Φ is satisfiable if and only if their exhaustive application does not introduce a contradiction. Here *contradiction* means the inference of both an atom ϕ and its negation $\neg\phi$. The IRs are presented in Fig. 3. For an IR r, the *antecedents* of r are the literals appearing above the line, while the *consequents* are those appearing below the line. We say that an IR r is *applicable* (to Φ) if there are terms appearing in Φ such that when these terms are substituted for the term placeholders of r (i.e. x, y, z, x_1, etc.), *all* of r's antecedents appear in Φ, and *none* of r's consequents appear in Φ.

We now explain each IR of Fig. 3. IDENT states that each node variable is equal to itself, while REFLEX enforces that any node variable is reachable from itself. TRANS1 states that the transitive closure f^* must extend the function f. TRANS2 simply enforces that f^* is transitive. FUNC asserts that if $f(x)=y$ and z is reachable from x, then z must also be reachable from y, unless $x=z$. If there is a cycle of length $k \geq 1$ in f, then it

follows that any node y reachable from a node on the cycle must be on the cycle as well; this is formalized by CYCLE$_k$. Similar to FUNC is SCC, which states that if x and y are distinct and mutually reachable from each other, and z is reachable from x, then x is reachable from z (since x must lie on a cycle of f). TOTAL requires that if y and z are both reachable from another node x, then there must exist some reachability relationship between y and z. The fact that in a cycle of f, no two distinct nodes x and y can have $f(x) = f(y)$ is captured by SHARE. Given the preceding intuition, it is easy to prove the following.

Theorem 1. *The inference rules of Fig. 3 are sound.*

Theorem 1 tells us that if iterative application of the IRs yields a contradiction, then we can conclude that the original set of literals is unsatisfiable. Conversely, we have proven our IRs to be complete with respect to sets of literals in a certain normal form, and Theorem 2 below states that it is sufficient to restrict attention to such sets. Let $\mathsf{Vars}(\Phi)$ denote the subset of the node variables V appearing in Φ.

Definition 1 (normal). *A set of literals Φ is said to be* normal *if*

1. *For each $v_i \in \mathsf{Vars}(\Phi)$, there exists*
   (a) at most one equality literal of the form $f(v_i) = v_j$, where $v_j \in \mathsf{Vars}(\Phi)$, and
   (b) the literal $v_i = v_i$.
   All equality literals of Φ are required to be of one of the forms (a) or (b).
2. *All inequality literals are of the form $\neg v_i = v_j$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
3. *All reachability literals are of the form $f^*(v_i, v_j)$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
4. *All unreachability literals are of the form $\neg f^*(v_i, v_j)$, where $v_i, v_j \in \mathsf{Vars}(\Phi)$.*
5. *There exist no data or boolean variable literals in Φ.*

Theorem 2. *There exists a polynomial-time algorithm that transforms any set Φ into a normal set Φ' such that Φ' is satisfiable if and only if Φ is satisfiable.*

Thanks to Theorem 2, our decision procedure can without loss of generality assume that Φ is normal. Let us call a set of literals Φ *consistent* if it does not contain a contradiction, and call Φ *closed* if none of the IRs of Fig. 3 are applicable. The following completeness result is the crux of our decision procedure.

Theorem 3. *If Φ is consistent, closed, and normal, then Φ is satisfiable.*

The proof of Theorem 3 is quite technical, and involves reasoning about the dependencies between digraphs of partial functions and the digraphs of their transitive closures. For details, please see our technical report [6].

Viewed from a high level, our decision procedure first applies the transformation of Theorem 2, and then repeatedly searches for an applicable IR, applies it (i.e. adds a consequent to the set), and recurses. The recursion is necessary for those IRs that *branch*, i.e. have multiple consequents. If the procedure ever infers a contradiction, it backtracks to the last branching IR with an unexplored consequent, or returns *unsatisfiable* if there is no such IR. If the procedure reaches a point where there are no applicable IRs and no contradictions, then the inferred set of literals is consistent, closed, and normal. Hence, by Theorem 3, it may correctly return *satisfiable*. Our technical report [6] provides a more formal presentation of the decision procedure. We note that our decision procedure is guaranteed to terminate because none of the IRs introduce new terms.

6.1 An Extension

In order to handle program assignments that mutate the links in the heap, i.e. modify f, we must extend our logic and decision procedure to support simultaneous reference to f and f', which respectively model the link function before and after the assignment. Such an assignment has the general form $f(\tau_1) := \tau_2$, where τ_1 and τ_2 are arbitrary terms. Lines 5 and 6 of the HMP of Fig. 1 are examples of such assignments. The semantic relationship between f and f' can be expressed using the well-known update operator:[7]

$$\Theta^e(f') = \mathtt{update}(\Theta^e(f), \Theta^e(\tau_1), \Theta^e(\tau_2)) \tag{2}$$

Rather than support update as an interpreted second order function symbol in the logic, we add inference rules that implicitly enforce the constraint (2). For each of the eight IRs of Fig. 3 that mention f, we add an analogous IR with f replaced with f'; these enforce analogous constraints between f'^*, f', and $=$ as are enforced by the unmodified IRs of Fig. 3 between f^*, f, and $=$. Furthermore, to enforce the constraint (2), the seven IRs of Fig. 4 are also included. The IRs introduce a fresh variable w that is forced to be equal to $f(\tau_1)$. This allows us to state that $\Theta^e(f) = \mathtt{update}(\Theta^e(f'), \Theta^e(\tau_1), \Theta^e(w))$, and hence the symmetry between the IRs UPDFUNC1 and UPDFUNC2, between UPDTRANS1 and UPDTRANS2, and between UPDTRANS3 and UPDTRANS4. Note that these IRs can introduce new terms, however, given a normal set of literals, the number of new terms is bounded. This implies that the extended decision procedure always terminates.

$$\frac{}{\begin{array}{l} f'(\tau_1) = \tau_2 \\ f(\tau_1) = w \end{array}}\ \textsc{Update}$$

$$\frac{f(x) = y}{\begin{array}{l} x = \tau_1 \\ y = w \end{array} \qquad f'(x) = y}\ \textsc{UpdFunc1} \qquad \frac{f'(x) = y}{\begin{array}{l} x = \tau_1 \\ y = \tau_2 \end{array} \qquad f(x) = y}\ \textsc{UpdFunc2}$$

$$\frac{f^*(x,y)}{\begin{array}{l} f'^*(x,\tau_1) \\ f'^*(w,y) \end{array} \qquad f'^*(x,y)}\ \textsc{UpdTrans1} \qquad \frac{f'^*(x,y)}{\begin{array}{l} f^*(x,\tau_1) \\ f^*(\tau_2,y) \end{array} \qquad f^*(x,y)}\ \textsc{UpdTrans2}$$

$$\frac{f^*(x,\tau_1) \qquad f'^*(x,y)}{f^*(x,y) \qquad f'^*(\tau_1,y)}\textsc{UpdTrans3} \qquad \frac{f'^*(x,\tau_1) \qquad f^*(x,y)}{f'^*(x,y) \qquad f^*(\tau_1,y)}\textsc{UpdTrans4}$$

Fig. 4. The update inference rules, which are used to extend our logic to support a second function symbol f', with the implicit constraint $f' = \mathtt{update}(f, \tau_1, \tau_2)$, where τ_1 and τ_2 are fixed but arbitrary terms, and w is a fresh variable used to capture $f(\tau_1)$. Note that the rule UPDATE can introduce literals that violate normalcy (Def. 1) in the case that τ_1 or τ_2 are not variables. However, this can be remedied by the addition of a new variable and equality literal for each sub-term of τ_1 and τ_2.

[7] If g is a function, a is an element in g's domain, and b is an element in g's codomain, then $\mathtt{update}(g,a,b)$ is defined to be the function $\lambda x.(\text{if } x = a \text{ then } b \text{ else } g(x))$.

Theorem 4. *The inference rules of Fig. 4 are sound.*

The proof of this theorem is provided in our technical report [6]. We have yet to flesh out the details of a proof of a conjecture analogous to Theorem 3 stating that this extended set of IRs is complete. However, we have empirical support for this conjecture: in conducting our experiments of Sect. 7, we never found any property violations caused by the extended decision procedure erroneously concluding that a set of literals was satisfiable. Of course, not having such a theorem *does not* compromise the soundness of verification by predicate abstraction.

7 Experiments

We have tested our tool on a number of HMP examples and summarized the results in Table 1. We ran the experiments on a Pentium 4 2.6 GHz machine. The safety properties we checked (when applicable) at the end of the HMP are:

- *no leaks* (NL) – all nodes reachable from the head of the list at the beginning of the program are also reachable at the end of the program.
- *insertion* (IN) – a distinguished node that is to be inserted into a list is actually reachable from the head of the list, i.e. the insertion "worked".
- *acyclic* (AC) – the final list is acyclic, i.e. nil is reachable from the head of the list.
- *cyclic* (CY) – list is a singly linked circular list, i.e. the head of the list is reachable from its successor.
- *sorted* (SO) – list is a sorted linked list, i.e. each node's data field is less than or equal to its successor's.
- *remove elements* (RE) – for examples that remove node(s), this states that the node(s) was (were) actually removed. For the program REMOVE-ELEMENTS, RE also asserts that the data field of all removed elements is false.

Often, the properties one is interested in verifying for HMPs involve universal quantification over the heap nodes. For example, to assert the property NL, we must express that for all nodes t, if t is reachable from *head* initially, then t is also reachable from *head* (or some other node) at the end of the program. Since our logic doesn't support quantification, we use the trick of introducing a Skolem constant t [15, 2] to represent a universally quantified variable. Here, t is a new node variable that is initially assumed to satisfy the antecedent of our property, and is otherwise unmodified by the program. For the example program of Fig. 1, we can express NL by conjoining $\neg t = \mathsf{nil} \wedge f^*(head, t)$ to the **assume** statement on line 2, and conjoining $f^*(head, t)$ to the assertion on line 12. Since t can be any non-nil node reachable from *head*, if the assertion is never violated, we have proven NL.

Our example programs are the following:

LIST-REVERSE – a classical HMP example that performs in-place reversal of a linked list.

LIST-ADD – a linked list is traversed, and the end of the list is reached. Then, a node is added to the end of the list.

ND-INSERT – pseudocode for this example is given in Fig. 1.

Table 1. Results of verifying HMPs. "property" specifies the verified property; "CFG edges" denotes the number of edges in the control-flow graph of the program; "preds" is the number of predicates required for verification; "time" is the average execution time over five runs to prove the properties; "DP calls" is the number of decision procedure queries. The largest memory usage for all these examples was 125 MB.

program	property	CFG edges	preds	time (sec)	DP calls
LIST-REVERSE	NL	6	8	1.1	173
LIST-ADD	$NL \wedge AC \wedge IN$	7	8	0.8	66
ND-INSERT	$NL \wedge AC \wedge IN$	5	13	7.9	258
ND-REMOVE	$NL \wedge AC \wedge RE$	5	12	19.3	377
ZIP	$NL \wedge AC$	20	22	280.7	9185
SORTED-ZIP	$NL \wedge SO \wedge IN$	28	22	249.9	13760
SORTED-INSERT	$NL \wedge AC \wedge SO$	10	20	217.6	8476
BUBBLE-SORT	$NL \wedge AC$	21	24	204.6	5930
BUBBLE-SORT	$NL \wedge AC \wedge SO$	21	27	441.0	6324
REMOVE-ELEMENTS	$NL \wedge CY \wedge RE$	15	17	1263.7	26721

ND-REMOVE – similar to ND-INSERT, except that instead of inserting a node, a node is nondeterministically chosen and removed from the list.

ZIP – zips two linked lists, shuffling the elements of both list into one. Then, the tail of the longer list is appended to the resulting list. This example is taken from a paper by Jensen et al. [20].

SORTED-ZIP – joins the elements of two sorted lists into one, also sorted. Here the data elements are simply booleans, so "sorted" means that all nodes with false fields come before nodes with true fields.

SORTED-INSERT – inserts a node into a sorted linked list so that sortedness is preserved. This is a modification of the example from a technical report by Lahiri and Qadeer [23].[8]

BUBBLE-SORT – The bubble sort example sorts elements of a linked list using the bubble sort algorithm. It is taken from a paper by Balaban et al. [2]. The data fields on which we sort are again booleans.

REMOVE-ELEMENTS – removes from a circular list all elements whose data field is false.

Our technical report [6] provides pseudocode and lists the required predicates for these examples.

As Table 1 shows, we were successful in verifying interesting properties of many examples in reasonable amounts of time. Of special note is our verification of sortedness for BUBBLE-SORT. This example is from Balaban et al. [2]; because of the BDD blow-up inherent in their decision procedure, their tool spaced out for the small model bound necessary for sound verification [1]. In contrast, our trading of space for time appears to be quite advantageous here.

[8] To simplify things, they require that the input list starts with a dummy element whose data field value has to be less than all possible values of that data field. We don't have such requirements in our example, which makes it slightly more complicated.

Running time of TVLA on the bubble sort example is somewhat faster than ours, although they are using a slower machine [24]. The recent experimental results of Manevich et al. [27] are significantly faster, in spite of the fact they were executed on a slower machine. For most of their examples, however, they only verify the simple property of no null dereferences (they also verify cyclicity for two examples). We are verifying more complicated properties, for instance SO. Very recently, Loginov et al. [26] have used TVLA to fully automatically verify the bubblesort example.

For the two examples in common with Lahiri and Qadeer [23],[9] LIST-REVERSE and SORTED-INSERT, we are significantly faster at verifying the same properties, with respective speed-ups of 75× and 6×. It should be noted, however, that we used a slightly faster machine, and also that for SORTED-INSERT, our data fields are merely booleans, while theirs are the full integers.

8 Future Work and Conclusions

Despite the fact that this work is in its early stages, our experiments demonstrate its effectiveness for verification of heap-manipulating programs. There are many directions for future research, which are outlined here.

We have identified the following issues related to the expressiveness of the simple transitive closure logic presented in this paper:

- This paper only supports a single link function f, yet clearly many heap-manipulating programs involve multiple link fields.
- We have found that even minimal support for universally quantified variables (as in the logic of Balaban et al. [2]) would allow expression of common heap structure attributes. For example, the current logic cannot assert that two terms x and y point to disjoint linked lists; a single universally quantified variable would allow for this property (see Nelson [30–page 22]). We found that capturing disjointedness is necessary for verifying that LIST-REVERSE always produces an acyclic list; hence we were unable to verify this property.
- Another situation that cannot be characterized relates to term ordering in circularly linked lists. Suppose x, y, and z are terms in such a list; we would like to express that y does or does not "come between" x and z in the list. Nelson [31] and Manevich et al. [27] have previously recognized the importance of such properties.

We believe that our decision procedure can be enhanced to handle each of these three cases. A final expressiveness deficiency, that we see no immediate solution to, is the expression of more involved heap structure properties, in particular trees. Though our logic cannot capture "x points to a tree", we believe that it is possible that an extension could be used to verify simple properties of programs that manipulate trees, for example that there are no memory leaks.

We also plan on investigating how existing techniques for predicate discovery and more advanced predicate abstraction algorithms mesh with our decision procedure. Our approach appears to be very promising, despite the fact that we have yet to harness the recent innovations in these areas.

[9] We were unable to run our tool on four of Lahiri and Qadeer's [23] examples because we have yet to implement support for data field mutations.

Acknowledgement

We acknowledge our mutual supervisor Alan J. Hu for his support during this project and Ittai Balaban, Shuvendu Lahiri, and Shaz Qadeer for answering our questions; we also thank Shaz for suggesting this research problem to us.

References

1. I. Balaban, 2005. personal correspondence.
2. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, 2005.
3. T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 203–213, 2001.
4. T. Ball, A. Podelski, and S.K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.
5. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *European Symposium on Programming (ESOP)*, 1999.
6. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs, 2005. UBC Dept. Comp. Sci. Tech Report TR-2005-19, http://www.cs.ubc.ca/cgi-bin/tr/2005/TR-2005-19.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, August 1986.
8. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Conf. on Computer Aided Verification (CAV)*, pages 78 – 92, 2002.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Conf. on Computer Aided Verification (CAV)*, pages 154–169, 2000.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symp. on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
11. D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 310–323, 2003.
12. S. Das and D. L. Dill. Successive approximation of abstract transition relations,. In *IEEE Symp. on Logic in Computer Science (LICS)*, 2001.
13. S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2002.
14. S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Conf. on Computer Aided Verification (CAV)*, 1999.
15. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Symp. on Principles of Programming Languages (POPL)*, pages 191–202, 2002.
16. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conf. on Computer Aided Verification (CAV)*, 1997.
17. D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symp. on Principles of Programming Languages (POPL)*, pages 58–70, 2002.

19. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive closure logics. In *Workshop on Computer Science Logic (CSL)*, pages 160–174, 2004.
20. J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 226–236, 1997.
21. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
22. N. Klarlund and M. I. Schwartzbach. Graph types. In *Symp. on Principles of Programming Languages (POPL)*, pages 196–205, 1993.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists, 2005. Microsoft Research Tech Report MSR-TR-2005-97.
24. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Intl. Symp. on Software Testing and Analysis*, pages 26–38, 2000.
25. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS'00)*, pages 280–301, 2000.
26. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *Conf. on Computer Aided Verification (CAV)*, pages 519–533, 2005.
27. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 181–198, 2005.
28. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Conf. on Computer Aided Verification (CAV)*, pages 476–490, 2005.
29. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Conf. on Programming Language Design and Implementation (PLDI)*, pages 221–231, 2001.
30. G. Nelson. *Techniques for program verification*. PhD thesis, Stanford University, 1979.
31. G. Nelson. Verifying reachability invariants of linked structures. In *Symp. on Principles of Programming Languages (POPL)*, pages 38–47, 1983.
32. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Programming Languages and Systems*, 24(3):217–298, 2002.
33. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 530–545, 2004.

Monitoring Off-the-Shelf Components*

A. Prasad Sistla, Min Zhou, and Lenore D. Zuck

University of Illinois at Chicago
{sistla, mzhou, lenore}@cs.uic.edu

Abstract. Software is being developed from off-the-shelf third party components. The interface specification of such a component may be under specified or may not fully match the user requirement. In this paper, we address the problem of customizing such components to particular users. We achieve this by constructing a monitor that monitors the component and detects any bad behaviors.

Construction of such monitors essentially involves synthesizing safety properties that imply a given property that is obtained from the interface specifications of the component and the goal specification of the user. We present various methods for synthesizing such safety properties when the given property is given by an automaton or a temporal logic formula. We show that our methods are sound and complete. These results are extensions of the results given in [11].

1 Introduction

The process of constructing software is undergoing rapid changes. Instead of a monolithic software development within an organization, increasingly, software is being assembled using third-party components (e.g., JavaBeans, .NET, etc.). The developers have little knowledge of, and even less control over, the internals of the components comprising the overall system.

One obstacle to composing agents is that current formal methods are mainly concerned with "closed" systems that are built from the ground up. Such systems are fully under the control of the user. Hence, problems arising from ill-specified components can be resolved by a close inspection of the systems. When composing agents use "off-the-shelf" ones, this is often no longer the case. Out of consideration for proprietary information, or in order to simplify presentation, companies may provide *incomplete specifications*. Worse, some agents may have no description at all except one that can be obtained by experimentation. Even if the component is completely specified, it may not fully satisfy the user requirements. Despite either of the cases, i.e., being ill-specified or mismatched-matched, "off-the-shelf" components might still be attractive enough so that the designer of a new service may wish to use them. In order to do so safely, the designer must be able to deal with the possibility that these components may exhibit undesired or unanticipated behavior, which could potentially compromise the correctness and security of the new system.

* This work is supported in part by the NSF grants CCR-9988884 and CCR-0205365.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 222–236, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The main problem addressed in this paper is that of customizing ill-specified or slightly mismatched off-the-shelf components for a particular user. We assume that we are given the interface specification Φ_I of the off-the-shelf component and the goal specification Φ which denotes the user requirement. We want to design a module M which runs in parallel with the off-the-shelf component and monitors its executions. If the execution violates the user specification Φ then the monitor M indicates this so that corrective action may be taken. Our customization only consists of *monitoring* the executions. (Here we are assuming that violation of Φ is not catastrophic and can be remedied by some other means provided it is detected in time; for example, leakage of the credit card number in a business transaction can be handled by alerting the credit card company.) See [11] for a motivating example.

Our goal is to obtain a specification ϕ for the module M so that M composed with the off-the-shelf component implies Φ. Further more, we want ϕ to be a safety property since violations of such properties can be monitored. Once such a specification ϕ is obtained as an automaton it is straight forward to construct the monitor M. Essentially, M would run the automaton on the executions of the off-the-shelf component and detect violations. Thus, given Φ_I and Φ, our goal is to synthesize a safety property ϕ so that $\Phi_I \wedge \phi \rightarrow \Phi$, or equivalently $\phi \rightarrow (\neg\Phi_I \vee \Phi)$, is a valid formula.

We considered the above problem in our previous work [11]. In that work, we concentrated on obtaining ϕ as a deterministic Büchi automaton. There we showed that while there is always some safety property ϕ that guarantees $\phi \rightarrow (\neg\Phi_I \vee \Phi)$ (e.g., the trivially false property), in general, there is no "maximal" such safety property. We also synthesized a family of safety properties ϕ_k, such that the higher k is, the more "accurate" and costly to compute ϕ_k is. We also defined a class of, possibly infinite state, deterministic *bounded automata*, that accept the desired property ϕ. For these automata we proved a *restricted completeness* result showing that if $(\neg\Phi_I \vee \Phi)$ is specified by a deterministic Büchi automaton $\mathcal{A}$ and $L(\mathcal{A})$ is the language accepted by $\mathcal{A}$ then for every safety property S contained in $L(\mathcal{A})$ there exists a bounded automaton that accepts the S. (Actually, the paper [11] erroneously stated that this method is complete in general; however, this was corrected in a revised version [12] claiming only restricted completeness.)

In this paper we extend these earlier results as follows. We consider the cases where $\neg\Phi_I \vee \Phi$ is given as an automaton or as a LTL formula. For the former case, when $\neg\Phi_I \vee \Phi$ is described by an automaton, we describe two ways of synthesizing the safety properties as a Büchi automaton $\mathcal{B}$. The first method assumes that the given automaton $\mathcal{A}$ is a non-deterministic Büchi automaton and constructs a non-deterministic $\mathcal{B}$ from $\mathcal{A}$ by associating a counter with each state. The constructed automaton is much simpler than the one given in [11]. We also define a class of infinite state automata and show that all these automata accept safety properties contained in $L(\mathcal{A})$. We also prove a restricted completeness result for this case.

In the second method we assume that $\mathcal{A}$ is given as a deterministic Streett automaton. For this case, we give the construction of a class of possibly infinite state automata that accept safety properties contained in $L(\mathcal{A})$. This construction employs a pair of counters for each accepting pair in the accepting condition of $\mathcal{A}$. We show that this construction is sound and is also complete when $\mathcal{A}$ is deterministic. Since deterministic Streett automata are more powerful than deterministic Büchi automata, this method is provably more powerful than the one given in [11]. Also, we can obtain a complete method for synthesizing safety properties contained in the language of a given non-deterministic Büchi or Streett automaton by systematically converting it into an equivalent deterministic Streett automaton [14] and by employing the above method on the resulting automaton.

In the case that Φ_I, Φ, and hence $\neg\Phi_I \vee \Phi$, are given as LTL formulas, we give semantic and syntactic methods. The semantic method constructs the tableau, from which it constructs a non-deterministic Büchi automaton that accepts the desired safety property. The syntactic method converts the formula $\neg\Phi_I \vee \Phi$ into another formula that specifies a safety property which implies $\neg\Phi_I \vee \Phi$.

Outline. Section 2 contains definitions, notation, and outlines some prior results relevant to this work. Section 3 studies synthesis of safety from specifications given by non-deterministic Büchi automata and shows a partial completeness result. Section 4 studies synthesis of safety from specifications given by deterministic Streett automata and shows a completeness result. Section 5 studies synthesis of safety from specifications given by LTL formulae. Section 6 discusses related literature, and we conclude in Section 7.

2 Preliminaries

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. If α_1 is a finite sequence and α_2 is a either a finite or a ω-sequence then $\alpha_1\alpha_2$ denotes the concatenation of the two sequences in that order.

For integers i and j such that $0 \leq i \leq j < |\sigma|$, $\sigma[i,j]$ denotes the (finite) sequence $s_i, \ldots s_j$. A *prefix* of σ is any $\sigma[0,j]$ for $j < |\sigma|$. We denote the set of σ's prefixes by $\mathit{Pref}(\sigma)$. Given an integer i, $0 \leq i < |\sigma|$, we denote by $\sigma^{(i)}$ the *suffix* of σ that starts with s_i.

For an infinite sequence $\sigma : s_0, \ldots$, we denote by $\inf(\sigma)$ the set of S-elements that occur in σ infinitely many times, i.e., $\inf(\sigma) = \{s : s_i = s$ for infinitely many i's$\}$.

Languages. A *language* L over a finite alphabet Σ is a set of finite or infinite sequences over σ. When L consists only of infinite strings (sequences), we sometimes refer to it as an *ω-language.* For a language L, we denote the set of prefixes of L by $\mathit{Pref}(L)$, i.e.,

$$\mathit{Pref}(L) = \bigcup_{\sigma \in L} \mathit{Pref}(\sigma)$$

Following [6, 2], an ω-language L is a *safety property* if for every $\sigma \in \Sigma^\infty$:

$$Pref(\sigma) \subseteq Pref(L) \quad \Longrightarrow \quad \sigma \in L$$

i.e., L is a safety property if it is *limit closed* – for every ω-string σ, if every prefix of σ is a prefix of some L-string, then σ must be an L-string.

Büchi Automata. A *Büchi automaton* (NBA for short) $\mathcal{A}$ on infinite strings is described by a quintuple $(Q, \Sigma, \delta, q^0, F)$ where:

- Q is a finite set of states;
- Σ is a finite alphabet of symbols;
- $\delta\colon Q \times \Sigma \to 2^Q$ is a transition function;
- $q^0 \in Q$ is an initial state;
- $F \subseteq Q$ is a set of accepting states.

The *generalized transition function* $\delta^*\colon Q \times \Sigma^* \to 2^Q$ 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* Büchi automaton (or DBA for short).

Let $\sigma :\ a_1, \ldots$ be an infinite sequence over σ. A *run* r *of* $\mathcal{A}$ on σ is an infinite sequence $q^0, \ldots$ over Q such that:

- $q^0 = q_0$;
- for every $i > 0$, $q^i \in \delta(q^{i-1}, a_i)$;

A run r on a Büchi automaton is *accepting* if $\inf(r) \cap F \neq \emptyset$. The automaton $\mathcal{A}$ *accepts* the ω-string σ if it has an accepting run over σ (for the case of DBAs, the automaton has a single run over σ). 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 an ω-language that is accepted by some (possibly non-deterministic) Büchi automaton.

A Büchi automaton $\mathcal{A}$ can also be used to define a *regular automaton* that is just like $\mathcal{A}$, only the acceptance condition of a run r is that its last state is accepting. We denote the regular language accepted by the regular version of $\mathcal{A}$ by $L_f(\mathcal{A})$.

Infinite-state. Büchi automata are defined just like Büchi automata, only that set of states may be infinite. We denote infinite-state DBAs by iDBAs, and infinite-state NBAs by iNBAs.

Streett Automata. A *Streett automaton* $\mathcal{S}$ on infinite strings is described by a quintuple $(Q, \Sigma, \delta, q^0, F)$ where Q, Σ, δ, and q^0 are just like in Büchi automata, and F is of the form $\cup_{i=1}^{m}(U_i, V_i)$ where each $U_i, V_i \subseteq Q$. A run r of $\mathcal{S}$ on $\sigma = a_1, \ldots$ is defined just like in the case of Büchi automaton. The run is *accepting* if, for every $i = 1, \ldots, m$, if $\inf(r) \cap U_i \neq \emptyset$ then $\inf(r) \cap V_i \neq \emptyset$, i.e., if some U_i states appears infinitely often in r, then some V_i states should also appear infinitely often in r.

Every Büchi automaton can be converted into a deterministic Streett automaton that recognizes the same ω-language ([14]).

Linear-time Temporal Logic. We consider LTL formulae over a set of atomic propositions Π using the boolean connectives and the temporal operators $\bigcirc$ (*next*), $\mathcal{U}$ (*until*), and $\mathcal{W}$ (*weak until* or *unless*). Let $\Sigma = 2^{\Pi}$. We define a satisfiability relation $\models$ between infinite sequences over Σ by:

For a proposition $p \in \Pi$, $\sigma \models p$ iff $p \in \sigma^0$
$\sigma \models \phi \vee \psi$ iff $\sigma \models \phi$ or $\sigma \models \psi$
$\sigma \models \neg\phi$ iff $\sigma \not\models \phi$
$\sigma \models \bigcirc\, \phi_1$ iff $\sigma^{(1)} \models \phi_1$
$\sigma \models \phi_1 \,\mathcal{U}\, \phi_2$ iff for some $i \geq 0$, $\sigma^{(i)} \models \phi_2$, and for all j, $0 \leq j < i$, $\sigma^{(j)} \models \phi_1$
$\sigma \models \phi_1 \,\mathcal{W}\, \phi_2$ iff for some $i \geq 0$, $\sigma^{(i)} \models \phi_1 \wedge \phi_2$, and for all j, $0 \leq j < i$, $\sigma^{(j)} \models \phi_1$

Note that the semantics of the unless operator is slightly different than the usual one, in requiring the ϕ_1 to hold on the state where ϕ_2 is first encountered.

For every LTL formula ϕ, we denote the set of atomic propositions that appear in ϕ by $Prop(\phi)$, and the ω-language that ϕ defines, i.e., the set of infinite sequences (models) that satisfy ϕ by $L(\phi)$.

Let ϕ be an LTL formula. We define the *closure* of ϕ, denoted by $cl(\phi)$ to be the minimal set of formulae that is closed under negation and includes ϕ, every subformula of ϕ, and for every subformula $\phi_1 \,\mathcal{W}\, \phi_2$ of ϕ the formula $\neg\phi_2 \,\mathcal{U}\, \neg\phi_1$. The *atoms* of ϕ, denoted by $at(\phi)$, is a subset of $2^{cl(\phi)-\emptyset}$ such that each atom $A \in at(\phi)$ is a maximally consistent subset of $cl(\phi)$ where for every $\psi = \phi_1 \,\mathcal{W}\, \phi_2 \in cl(\phi)$, $\psi \in A$ iff $(\neg\phi_2 \,\mathcal{U}\, \neg\phi_1) \notin A$. An *initial atom* is any atom that contains ϕ. The *tableau* of ϕ, $tab(\phi)$, is a graph $(at(\phi), R)$ whose nodes are $at(\phi)$, and a $(A_1, A_2) \in R$ iff the following all hold:

For every $\bigcirc\, \psi \in cl(\phi)$, $\bigcirc\, \psi \in A_1$ iff $\psi \in A_2$
For every $\psi_1 \,\mathcal{U}\, \psi_2 \in cl(\phi)$, $\psi_1 \,\mathcal{U}\, \psi_2 \in A_1$ iff $\psi_2 \in A_1$ or $\psi_1 \in A_1$ and $\psi_1 \,\mathcal{U}\, \psi_2 \in A_2$
For every $\psi_1 \,\mathcal{W}\, \psi_2 \in cl(\phi)$, $\psi_1 \,\mathcal{W}\, \psi_2 \in A_1$ iff $\psi_1, \psi_2 \in A_1$ or $\psi_1, \psi_1 \,\mathcal{W}\, \psi_2 \in A_2$

It is known (e.g., [9]) that ϕ is satisfiable iff $tab(\phi)$ contains a path leading from an initial atom into a maximally strongly connected component (MSCC) $\mathcal{C}$ such that for every $\psi_1 \,\mathcal{U}\, \psi_2 \in A \in \mathcal{C}$, there is an atom $B \in \mathcal{C}$ such that $\psi_2 \in B$.(Such MSCCs are called “fulfilling.”) Similarly, it is also known (see, e.g., [18, 3]) how to construct a NBA (and, consequently, a deterministic Streett automaton) that recognizes $L(\phi)$.

3 Synthesizing Safety from Büchi Automata

In this section we study synthesis of safety properties from a given NBA. We fix a (possibly non-deterministic) Büchi automaton $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_A^0, F_A)$. As shown in [11], unless $L(\mathcal{A})$ is already safety, there is no maximal safety property that is contained in $L(\mathcal{A})$ which is Büchi recognizable. We first show an infinite

chain of safety properties that are all Büchi recognizable, each contained in the next. We then present *bounded Büchi automaton*, an infinite-state version of NBAs, and show that each accepts a safety property in $L(\mathcal{A})$. We also show a partial completeness result, namely, that if $\mathcal{A}$ is deterministic then each safety property contained in $L(\mathcal{A})$ is accepted by some bounded automata.

Both constructions, of the chain of Büchi automata and the bounded automata, are much simplified versions of their counterparts described in [11].

3.1 Synthesis of Safety into Büchi Automata

We define a chain of safety properties $\{L_k(\mathcal{A})\}_{k>0}$ such that for every k, $L_k(\mathcal{A}) \subseteq L(\mathcal{A})$ and $L_k(\mathcal{A}) \subseteq L_{k+1}(\mathcal{A})$.

Let $k > 0$ be an integer. The ω-language $L_k(\mathcal{A})$ is a subset of $L(\mathcal{A})$, where every string has an accepting $\mathcal{A}$-run with the first accepting (F_A) state appearing within the first k states of the run, and any successive accepting states in the run are separated by at most k states. Formally, $w \in L_k(\mathcal{A})$ iff there exists an accepting $\mathcal{A}$ run $r_0, \ldots$ such that the set of indices $I = \{i \geq 0 : r_i \in F_A\}$ on w satisfying the following condition: for some $\ell < k$, $\ell \in I$, and for every $i \in I$, there is some $j \in I$ such that $i < j < i + k$.

For $k \geq 0$, let $\mathcal{B}_k\colon (Q_k\,, \Sigma\,, \delta_k\,, \langle q_A^0, k-1\rangle\,, Q_k)$ be a NBA where:

- $Q_k = Q_A \times \{0, 1, \cdots, k-1\}$
- $\langle q', i'\rangle \in \delta_k(\langle q, i\rangle, a)$ iff
  - $i > 0$, $q \notin F_A$, and $i' = i - 1$;
  - $i \geq 0$, $q \in F_A$, and $i' = k - 1$.

The automaton $\mathcal{B}_k$ simulates the possible runs of $\mathcal{A}$ on the input and uses a modulo k counter to maintain the number of steps within which an accepting state should be reached. Note that, when $q \notin F_A$, there are no outgoing transitions from the state $\langle q, 0\rangle$. Since all $\mathcal{B}_k$-states are accepting states, $L(\mathcal{B}_k)$ is a safety property. Also, from the construction it immediately follows that $L(\mathcal{B}_k) = L_k(\mathcal{A})$. We therefore conclude:

Lemma 1. *$L_k(\mathcal{A})$ is a safety property, and $L(\mathcal{B}_k) = L_k(\mathcal{A})$.*

3.2 Synthesis of Safety into Bounded Automata

The construction of the previous section is not complete in the sense that there are always safety properties in $L(\mathcal{A})$ that are not recognized by $\mathcal{A}_k$. We introduce bounded automata, a new type of Büchi automata, and show that (1) they only recognize safety properties in $L(\mathcal{A})$, and (2) when $\mathcal{A}$ is deterministic, they can recognize every safety property contained in $L(\mathcal{A})$.

Assume some (possible infinite) set Y. A *bounded automaton over $\mathcal{A}$ using Y* is a (i)NBA described by a tuple $\mathcal{N}\colon (Q_N, \Sigma, \delta_N, q_N^0, F_N)$ where:

- $Q_N \subseteq Y \times Q_A \times (\mathbb{N} \cup \{\infty\})$. Given a state $q_N = \langle r, q, i\rangle \in Q_N$, we refer to r as the Y component of q_N, to q as the $\mathcal{A} -$ *state* of q_N, and to i as the *counter* of q_N;

- For every $\langle r, q, i\rangle \in Q_N$, if $(q', r', i') \in \delta_N(\langle q, r, i\rangle, a)$ then the following all hold:
  - $q' \in \delta_A(q, a)$;
  - If $i = \infty$ then $i' = \infty$;
  - If $q \notin F_A$, then $i' < i$ or $i' = \infty$;
- q_N^0 is in $Y \times \{q_A^0\} \times \mathbb{N}$;
- $F_N = Y \times Q_A \times \mathbb{N}$.

Note that there are no outgoing from a state whose $\mathcal{A}$-state is not in F_A and whose counter is 0. Also, once a run reaches a state with counter value ∞, it remains in such states. Since the counters of states with non-accepting $\mathcal{A}$-states either decrease or become ∞, it follows that once a run reaches a rejecting state (i.e., a state in $Q_N \setminus F_N$), it remains there. It thus follows from [16] that $L(\mathcal{N})$ is a safety property. Consider an accepting $\mathcal{N}$ run. Since the counters of states can only decrease finitely many times from non-accepting $\mathcal{A}$-states, it follows that the run has infinitely many accepting $\mathcal{A}$-states. Thus, its projection on the $\mathcal{A}$-states is an accepting $\mathcal{A}$-run. We can therefore conclude:

Lemma 2. *For a bounded automaton $\mathcal{N}$ over $\mathcal{A}$, $L(\mathcal{N})$ is a safety property in $L(\mathcal{A})$.*

Lemma 2 shows that bounded automata over $\mathcal{A}$ accept only safety properties that are contained in $L(\mathcal{A})$. We next identify the safety properties in $L(\mathcal{A})$ that bounded automata accept.

Recall that for a Büchi automaton $\mathcal{A}$, $L_f(\mathcal{A})$ is the regular language defined by the regular version of $\mathcal{A}$. Let $S \subseteq L(\mathcal{A})$ be a safety property. Similarly to [11], we define:

- For a sequence $\sigma \in S$ and $\alpha \in \mathit{Pref}(\sigma)$, let $\mathit{min_idx}(\sigma, \alpha) = \mathit{min}\{|\beta| : \alpha\,\beta \in L_f(\mathcal{A}) \cap \mathit{Pref}(\sigma)\}$;
- For any $\alpha \in \Sigma^*$, let $Z(\alpha, S) = \{\mathit{min_idx}(\sigma, \alpha) : \sigma \in S \wedge \alpha \in \mathit{Pref}(\sigma)\}$

Note that if $\alpha \in L_f(\mathcal{A}) \cap \mathit{Pref}(\sigma)$, then $\mathit{min_idx}(\alpha, a) = 0$ and $Z(\alpha, S) = \{0\}$. Similarly, if $\alpha \notin \mathit{Pref}(S)$, then $Z(\alpha, S) = \emptyset$. It is shown in [11] that for every $\alpha \in \Sigma^*$, $Z(\alpha, S)$ is finite.

For any $\alpha \in \Sigma^*$, we define

$$\mathit{idx}(\alpha, S) = \begin{cases} \max_{i \in Z(\alpha,S)}\{i\} & Z(\alpha, S) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

Thus, $\mathit{idx}(\alpha, S) \in \mathbb{N}$ iff $\alpha \in \mathit{Pref}(S)$.

Let $S \subseteq L(\mathcal{A})$ be a safety property. A *bounded S-automaton over $\mathcal{A}$* is a bounded automaton over $\mathcal{A}$ using Σ^*, of the form $\mathcal{D}$: $(Q_D, \Sigma, \delta_D, q_D^0, F_D)$, where:

- $Q_D = \{\langle \alpha, q, i\rangle : \alpha \in \Sigma^*,\ q \in Q_A,\ i \in \{\mathit{idx}(\alpha, S)\, \infty\}\}$;
- For every $q \notin F_A$, if $\langle \alpha', q', i'\rangle \in \delta_D(\langle \alpha, q\, i\rangle, a)$ then the following all hold:
  1. $\alpha' = \alpha\, a$;
  2. $q' \in \delta_A(q, a)$;

3. $i' = \begin{cases} idx(\alpha\, a, S) \text{ if } i \neq \infty \text{ and } idx(\alpha\, a, S) < i \\ \infty \qquad\qquad \text{otherwise} \end{cases}$

– $q_D^0 = \langle \epsilon, q_A^0, idx(\epsilon, S) \rangle$.
– $F_D = \Sigma^* \times Q_A \times \mathbb{N}$

Theorem 1 (Partial Completeness). *For a safety property $S \subseteq L(\mathcal{A})$, $L(\mathcal{D}) \subseteq S$, and if $\mathcal{A}$ is a deterministic automaton then $L(\mathcal{D}) = S$.*

Proof. For (1), We show that for any $\sigma \in \Sigma^\omega$, if $\sigma \notin S$, then $\sigma \notin L(\mathcal{D})$. Assume $\sigma \in \Sigma^\omega \setminus S$. Since S is a safety property, there exists an integer i such that every prefix $\alpha \prec \sigma$, of length i or more, $\alpha \notin \mathit{Pref}(S)$. Consider such a prefix α. Obviously, $idx(\alpha, S) = \infty$. Hence, in any run, after reading α, the counter of the state reached is ∞. It follows that $\sigma \notin L(\mathcal{D})$.

For (2), assume that $\sigma \in S$. Define a sequence $\{p_i\}_{i \geq 0}$ over Σ^* such that $p_0 = \epsilon$ and for every $i > 0$, $p_i = \sigma[0, i-1]$, i.e., $\{p_i\}_{i \geq 0}$ is the sequence of σ's prefices. Since $S \subseteq L(\mathcal{A})$, there exists an accepting $\mathcal{A}$-run $r_A^0, \cdots$ of $\mathcal{A}$ on σ. Consider now the sequence of $\mathcal{D}$ states $\alpha = \{\langle p_j, r_j^A, idx(p_j, S) \rangle\}_{j \geq 0}$. The first element in α is q_D^0. Consider now the case that $\mathcal{A}$ is deterministic. When the $\mathcal{A}$ state of an α-state is $\mathcal{A}$-accepting, its counter is 0; otherwise, the counter of the next α-state is lower. Thus, α is a $\mathcal{D}$-run. Finally, since $\sigma \in S$, every counter in α is non-∞, thus $\mathcal{D}$ is accepting. □

To see why the method is incomplete for general NBAs, consider the NBA $\mathcal{A}$ described in Figure 1. Obviously, $L(\mathcal{A}) = L_1 \cup L_2$ where $L_1 = \{a^i b \Sigma^\omega : i > 0\} \cup \{a^\omega\}$ and $L_2 = \{\Sigma^i c^\omega : i \geq 0\}$. Note that L_1 is generated by the sub-automaton consisting of $\{q_0, q_1, q_2, q_3\}$ and L_2 is generated by the sub-automaton consisting of $\{q_0, q_4, q_5\}$. The language L_1 is clearly a safety property, however, the above method cannot generate L_1, since any bounded automaton where the value of the counter in the initial state is k can only accept the L_1 strings of the form $\{a^i b \Sigma^\omega : 0 < i \leq k\} \cup \{a^\omega\}$.

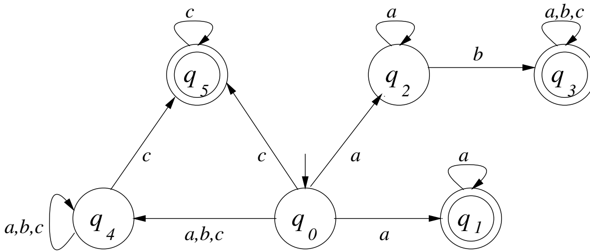


Fig. 1. Automaton $\mathcal{A}$

4 Synthesizing Safety from Streett Automata

In this section, we generalize the construction of the bounded automata of the previous section into *extended bounded automata* for synthesizing safety properties contained in the language of a given Streett automaton (SA). We show that

this construction is sound, and it is complete when the given automaton is a deterministic Streett Automaton (DSA). Since NBAs and SAs can be determinized into an equivalent deterministic Streett automata (DSAs) [14], the construction given in this section can be used to synthesize any safety property contained in the language accepted by a given NBA or SA, thus giving a complete system for these automata also.

Let $\mathcal{A} = (Q_A, \Sigma, \delta_A, q_A^0, F_A)$ be a SA. Assume that $F_A = \cup_{i=1}^{m}\{(U_i, V_i)\}$. Without loss of generality, assume that for every $i = 1, \ldots, m$, $U_i \cap V_i = \emptyset$. (If $U_i \cap V_i \neq \emptyset$, then U_i can be replaced by $U_i \setminus V_i$ without impacting the recognized language.)

Let Y be some (possibly infinite) set. An *extended bounded automaton* $\mathcal{E}$ *over* $\mathcal{A}$ *using* Y is a (i)NBA $(Q_E, \Sigma, \delta_E, q_E^0, F_E)$ where:

- $Q_E \subseteq Y \times Q_A \times (\mathbb{N} \cup \{\infty\})^{(2m)}$;
- For every $R = \langle r, q, i_1, j_1, \ldots, i_m, j_m\rangle \in Q_E$ and $a \in \Sigma$, if $R' \in \delta_E(R, a)$ where $R' = \langle r', q', i'_1, j'_1, \ldots, i'_m, j'_m\rangle$ the following all hold:
  - $q' \in \delta_A(q, a)$;
  - For every $k = 1, \ldots, m$:
    1. If $i_k = \infty$ then $i'_k = \infty$ and if $j_k = \infty$ then $j'_k = \infty$;
    2. if $q' \in U_k$ then, if $i_k > 0$ then $i'_k < i_k$, and if $i_k = 0$ then $i'_k = 0$ and $j'_k < j_k$.
    3. $q' \notin (U_k \cup V_k)$ then,if $i_k > 0$ then $i'_k \leq i_k$, and if $i_k = 0$ then $i'_k = 0$ and $j'_k \leq j_k$.
- $q_E^0 \in Y \times \{q_A^0\} \times \mathbb{N}^{(2m)}$;
- $F_E = Y \times Q_A \times \mathbb{N}^{(2m)}$.

Extended bounded automata are similar to bounded automata. However, states of extended bounded automata associate a pair of counters (i_k, j_k) for each accepting pair (U_k, V_k) in F_A, and the transition function is different. Just like in the case of bounded automata, when an extended automaton enters a rejecting state, it cannot re-enter an accepting state. It thus follows from [16] that extended bonded automata can only recognize safety properties.

Consider an accepting run $\rho = R_1, \ldots$ of $\mathcal{E}$, where for every $k \geq 0$, $R_k = \langle r_k, q_k, i_{k,1}, j_{k,1}, \ldots, i_{k,m}, j_{k,m}\rangle$. For $Q' \subseteq Q_A$ and $k \geq 0$, we say that a *Q'-state appears in* R_k if $q_k \in Q'$. Assume that for some $\ell = [1..m]$, U_ℓ appears infinitely many times in ρ's states, and let $k \geq 0$. Consider now the sequence of $i_{k\ ,\ell}$ for $k' \geq k$. Since ρ is accepting, and there are infinitely many R_k where a U_ℓ-state appears, $i_{k\ ,\ell}$ never increases until some V_ℓ-state appears, and decreases with each appearance of of a U_ℓ-state. Once $i_{k\ ,\ell}$ becomes zero, the value of $j_{k\ ,\ell}$ decrease with each appearance of a U_ℓ-state. Thus, a V_ℓ state must appear in ρ after R_k. It thus follows that R, projected onto its Q_A-states, is an accepting run of $\mathcal{A}$. We can therefore conclude:

Theorem 2 (Soundness). *For every extended bounded automaton $\mathcal{E}$ over $\mathcal{A}$, the language recognized by $\mathcal{E}$ is a safety property that is contained in $L(\mathcal{A})$.*

We now turn to prove completeness, i.e., we show that if $\mathcal{A}$ is deterministic then *every* safety property in $L(\mathcal{A})$ is recognized by some extended bounded

automaton. We fix some safety property $S \in L(\mathcal{A})$ and show it is recognized by some extended bounded automaton.

For the proof we define and prove some properties of (finitely branching) infinite labeled trees. For details on the definition and proofs see the complete version of this paper in [15]. For space reasons, we only outline the main ideas here.

A tree is *finitely branching* if each node has finitely many children. Consider a finitely branching tree T with a labeling function ℓ from T's nodes that labels each nodes with one of three (mutually disjoint) labels, $\texttt{lab}_1$, $\texttt{lab}_2$, and $\texttt{lab}_3$. An infinite path in T is *acceptable* with respect to ℓ over $(\texttt{lab}_1, \texttt{lab}_2, \texttt{lab}_3)$ if it either contains only finitely many nodes with $\texttt{lab}_1$-labels ($\texttt{lab}_1$-nodes), or it contains infinitely many nodes with $\texttt{lab}_2$ labels ($\texttt{lab}_2$-nodes). The tree T is acceptable with respect to ℓ if each of its infinite paths is, and it is acceptable if it is acceptable with respect to some labeling function as above.

A *ranking function* on T is a function associating each node with a non-negative integer. Pair (ρ_1, ρ_2) of ranking functions is *good* if for every two nodes n and n' in T such that n is the parent of n', the following hold:

1. If n' is a $\texttt{lab}_1$-node and $\rho_1(n) > 0$ then $\rho_1(n) > \rho_1(n')$.
2. If n' is a $\texttt{lab}_1$-node and $\rho_1(n) = 0$ then $\rho_1(n') = 0$ and $\rho_2(n) > \rho_2(n')$.
3. If n' is a $\texttt{lab}_3$-node and $\rho_1(n) > 0$ then $\rho_1(n) \geq \rho_1(n')$.
4. If n' is a $\texttt{lab}_3$-node and $\rho_1(n) = 0$ then $\rho_1(n') = 0$ and $\rho_2(n) \geq \rho_2(n')$.

Theorem 3. **[15]** *A labeled finitely-branching tree T is acceptable iff there is a good pair of ranking functions for it.*

We can now prove the completeness theorem:

Theorem 4. *Let $\mathcal{A}$ be a deterministic Streett automaton and $S \subseteq L(\mathcal{A})$ be a safety property. There exists an extended bounded automaton $\mathcal{B}$ such that $L(\mathcal{B}) = S$.*

Proof. Consider the finitely branching tree T whose set of nodes is $\{(\alpha, q) : \alpha \in \mathit{Pref}(S),\ q = \delta^*_A(q^0_A, \alpha)\}$, its root is (ϵ, q^0_A), and for any two nodes $n = (\alpha, q)$ and $n' = (\alpha', q')$, n' is a child of n iff for some $a \in \Sigma$, $\alpha' = \alpha\ a$ and $q' \in \delta_A(q, a)$. Then, for an infinite path π starting from the root, its projection on its second component is an accepting run of $\mathcal{A}$ on the string which is the limit of its first projection (on $\mathit{Pref}(S)$).

For every $k = 1, \ldots, m$, let ℓ_k be a labeling of T by the labels $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{n}$ such that every U_k node in T (i.e., a node whose second component is in U_k) is labeled with $\mathbf{u}$, every V_k node is labeled with $\mathbf{v}$, and every node that is in neither U_k or V_k is labeled with $\mathbf{n}$. It follows that T is acceptable with respect to the labeling ℓ_k over $(\mathbf{u}, \mathbf{v}, \mathbf{n})$.

It follows from Theorem 3 that for every $k = 1, \ldots, m$, there exists a pair $(\rho_{k,1}, \rho_{k,2})$ of ranking functions such that for every two nodes $n = (\alpha, q)$ and $n' = (\alpha', q')$ such that n is the parent of n', the following all hold:

– If n' is a U_k node then
  - If $\rho_{k,1}(n) > 0$ then $\rho_{k,1}(n') < \rho_{k,1}(n)$;
  - If $\rho_{k,1}(n) = 0$ then $\rho_{k,1}(n') = 0$ and $\rho_{k,2}(n') < \rho_{k,2}(n)$;

– If n' is neither a U_k nor a V_k node, then
  - If $\rho_{k,1}(n) > 0$ then $\rho_{k,1}(n') \leq \rho_{k,1}(n)$;
  - If $\rho_{k,1}(n) = 0$ then $\rho_{k,1}(n') = 0$ and $\rho_{k,2}(n') \leq \rho_{k,2}(n)$;

We now define an extended bounded automaton $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_E^0, F_E)$ where:

– For every $(\alpha, q, i_1, j_1, ..., i_m, j_m) \in Q_E$,
  1. $q = \delta_A^*(q_A^0, \alpha)$;
  2. If $(\alpha, q) \in T$ then, for each $k = 1, \ldots, m$, $i_k = \rho_{k,1}((\alpha, q))$ and $j_k = \rho_{k,2}((\alpha, q))$;
  3. If $(\alpha, q) \notin T$ then, for each $k = 1, \ldots, m$, $i_k = j_k = \infty$. Note that this definition guarantees that for every finite string α and $\mathcal{A}$-state q, there is a unique $\mathcal{E}$-state whose first two coordinates are α and q.

– For every state $R = (\alpha, q, \ldots) \in Q_E$ and symbol $a \in \Sigma$, $\delta_E(R, a)$ is the unique Q_E state whose first two coordinates are $\alpha\, a$ and $\delta_A(q, a)$;
– q_E^0, the initial state of $\mathcal{E}$, is the unique state whose first two coordinates are ϵ and q_A^0;
– F_E is the set of states such that for each $k = 1, \ldots, m$, $i_k, j_k \neq \infty$.

From the properties of the ranking functions it now follows that $L(\mathcal{E}) = S$. □

5 Synthesizing Safety from Temporal Formulae

In Sections 3 and 4 we describe how to synthesize safety properties from NBAs and SAs. In this section we discuss how to directly synthesize a safety property from an LTL formula.

Let ϕ be a LTL formula. As discussed in Section 2, one can construct $tab(\phi)$ and then an NBA, or a DSA, that recognizes $L(\phi)$, from which any of the approaches described in Section 3 can be used. However, such approaches may drastically alter $tab(\phi)$. In this section we describe two methods to obtain safety properties from $tab(\phi)$ while preserving its structure. The first is semantics based, and consists of augmentation to $tab(\phi)$. The second is syntactic based, and uses $tab(\phi)$ for monitoring purposes.

5.1 A Semantic Approach

Let ϕ be an LTL formula and consider $tab(\phi) = (at(\phi), R)$. We construct an expanded version of the tableau. We outline the construction here and defer formal description to the full version of the paper:

With each formula $\psi = \psi_1\, \mathcal{U}\, \psi_2 \in cl(\phi)$ we associate a counter c_ψ that can be "active" or "inactive". (Thus, we split atoms into new states, each containing the atoms and the counters.) If $(A_1, A_2) \in R$, then in the new structure, if the counter associated with ψ in A_1 is non-zero and active, and $\psi_2 \notin A_2$, then the counter is decremented; if $\psi_2 \in A_2$, the counter becomes inactive. If the value of the counter is zero, the transition is disabled. If the counter is inactive and $\psi \in A_2$, then the counter becomes active and is replenished to some constant k.

An initial node of the expanded tableau is one which corresponds to an initial atom, where only counters of $\mathcal{U}$ formulae that are in the node are active (and set to k) and the others inactive. Obviously, a "good" path in the new structure is one that starts at an initial node, and for every $\mathcal{U}$ formula in $cl(\phi)$, either the counter of the formula is infinitely many often inactive, or it is infinitely often equal to k.

In the full version of the paper we will show how the new structure defines a NBA that accepts safety properties in $L(\phi)$.

5.2 A Syntactic Approach

Let ϕ be an LTL formula where all negations are at the level of propositions. This is achieved by the following rewriting rules:

$$\begin{array}{llll} \neg(\psi_1 \vee \psi_2) \Longrightarrow (\neg\psi_1 \wedge \neg\psi_2) & & \neg(\psi_1 \wedge \psi_2) \Longrightarrow (\neg\psi_1 \vee \neg\psi_2) \\ \neg \bigcirc \psi \Longrightarrow \bigcirc\neg\psi & & \neg(\psi_1 \,\mathcal{U}\, \psi_2) \Longrightarrow (\neg\psi_2 \,\mathcal{W}\, \neg\psi_1) \end{array}$$

$$\neg(\psi_1 \,\mathcal{W}\, \psi_2) \Longrightarrow (\neg\psi_2 \,\mathcal{U}\, \neg\psi_1)$$

Let k be a positive integer. We construct an LTL formula ϕ_k by replacing each sub-formula of the form $\psi_1 \,\mathcal{U}\, \psi_2$ appearing ϕ with $\psi_1 \,\mathcal{U}_{\leq k}\, \psi_1$ where $\mathcal{U}_{\leq k}$ is the *bounded until* operator (i.e., $\mathcal{U}_{\leq k}$ guarantees its right-hand-side within k steps.) The following theorem, which gives a syntactic method for synthesizing safety properties, can be proven by induction on the length of ϕ. The monitor for ϕ_k can thenb be built by obtaining its tableau.

Theorem 5. *Let ϕ be a temporal formula, let k be positive integer, and let ϕ_k be as defined above. Then $L(\phi_k)$ is a safety property which implies ϕ.*

6 Related Work

As indicated in the introduction, in [11, 12] we studied the problem of synthesizing safety from Büchi specifications and presented a solution that satisfies restricted completeness in the sense that not all safety property can be synthesized. The work here presents a solution that is both simpler and complete, namely, given a NBA $\mathcal{A}$, the construction here generates any safety property that is in $L(\mathcal{A})$. In addition, the work here presents synthesis of safety property directly from LTL properties. These methods are much simpler than the ones given in [11].

Similar in motivation to ours, but much different in the approach, is the work in [13]. There, the interaction between the module and the interface is viewed as a 2-player game, where the interface has a winning strategy if it can guarantee that no matter what the module does, Φ is met while maintaining Φ_I. The work there only considers determinisic Büchi automata. The approach here is to synthesize the interface behavior, expressed by (possibly) non-determinisic automata, before constructing the module.

Some of the techniques we employ are somewhat reminiscent of techniques used for verifying that a safety property described by a state machine satisfies a

correctness specification given by an automaton or temporal logic. For example, simulation relations/state-functions together with well-founded mappings [5, 1, 17] have been proposed in the literature for this purpose. Our bounded automata use a form of well-founded mappings in the form of positive integer values that are components of each state. (This is as it should be, since we need to use some counters to ensure that an accepting state eventually appears.) However, here we are not trying to establish the correctness of a given safety property defined by a state machine, but rather, we are deriving safety properties that are contained in the language of an automaton.

In [7, 8] Larsen et al propose a method for turning an implicit specification of a component into an explicit one, i.e., given a context specification (there, a process algebraic expression with a hole, where the desired components needs to be plugged in) and an overall specification, they fully automatically derive a temporal safety property characterizing the set of all implementations which, together with the given context, satisfy the overall specification. While this technique has been developed for component synthesis, it can also be used for synthesizing optimal monitors in a setting where the interface specification Φ_I and the goal specification Φ are both safety properties. In this paper, we do not make any assumptions on Φ_I and Φ. They can be arbitrary properties specified in temporal logic or by automata. We are aiming at exploiting liveness guarantees of external components (contexts), in order to establish liveness properties of the overall system under certain additional safety assumptions, which we can run time check (monitor). This allows us to guarantee that the overall system is as live as the context, as long as the constructed monitor does not cause an alarm.

There has been much on monitoring violations of safety properties in distributed systems. In these works, the safety property is typically explicitly specified by the user. Our work is more on deriving safety properties from component specifications than developing algorithms for monitoring given safety properties. In this sense, the approach to use safety properties for monitoring that have been automatically derived by observation using techniques adapted from automata learning (see [4]) is closer in spirit to the technique here. Much attention has since been spent in optimizing the automatic learning of the monitors [10]. However, the learned monitors play a different role: whereas the learned monitors are good, but by no means complete, sensors for detecting unexpected anomalies, the monitors derived with the techniques of this paper *imply* the specifying property as long as the guarantees of the component provider are true.

7 Conclusions and Discussion

In this paper, we considered the problem of customizing a given, off-the-shelf, reactive component to user requirements. In this process, we assume that the reactive module's external behavior is specified by a formula Φ_I and the desired goal specifications is given by a formula Φ. We presented methods for obtaining a safety specification ϕ so that $\phi \rightarrow (\neg\Phi_I \vee \Phi)$ by synthesizing (possibly infinite-state) NBAs for monitoring off-the-shelf components.

More specifically, we considered three different cases. The first one is when $(\neg\Phi_I \vee \Phi)$ is given as a non-deterministic Büchi automaton. For this case, the synthesized safety property is also given as a non-deterministic automaton. This method is shown to to be sound and complete when the given automaton is deterministic. This method is simpler than the one given in [11]. The second case is when $(\neg\Phi_I \vee \Phi)$ is given as a Streett automaton. In this case also, we gave a sound method for synthesizing safety properties contained in the language of the automaton, The method is also shown to be complete if the given automaton is a deterministic Streett Automaton. Since every Büchi automaton and Streett automaton can be converted algorithmically into an equivalent deterministic Streett automaton, this method gives us a complete system for synthesizing any safety property contained in the language of a given Büchi automaton or Streett automaton. The last case is when $\neg\Phi_I \vee \Phi$ is a LTL formula. In this case, we outlined a semantic method that works directly with tableux associated with formulae, without converting the tableux into automata. We also gave a syntactic method for this.

For our automata to be useful, they should be recursive, i.e., their set of states and their transition functions should be recursive functions. For monitoring purposes we need not explicitly compute the automaton and keep it in memmory, rather, we only need to maintain its current state and, whenever a new input arrives, we can use the transition function to compute the next state. For a non-dterministic automaton, we need to maintain the set of reachable states. Because of finite non-determinism, this set will be finite and is also computed on-the-fly after each input.

It is to be noted that the counters that are used only count down after occurrence of some input symbols. If the off-the-shelf component never responds, the autoamton remains in its current (good) state and permits such computations. This problem can be overcome by assuming that clock ticks are inputs to the automaton as well, and allowing a counter to count down with (some or all) clockticks. There are also other possible solutions to this problem.

We implemented a preliminary version of the method given in Section 3. It will be interesting to implement the general version given in Section 4 and apply it to practical problems. It will also be interesting to investigate how the counter values, given in these constructions, can be computed as functions of the history seen thus far. Real-time implementation of such systems need to be further investigated.

References

1. M. Abadi and L. Lamport. The existence of state mappings. In *Proceedings of the ACM Symposium on Logic in Computer Science*, 1988.
2. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
3. E. A. Emerson and A. P. Sistla. Triple exponential decision procedure for the logic CTL*. In *Workshop on the Logics of Program, Carnegie-Mellon University*, 1983.

4. H. Hungar and B. Steffen. Behavior-based model construction. *STTT*, 6(1):4–14, 2004.
5. B. Jonsson. Compositional verification of distributed systems. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, 1987.
6. L. Lamport. Logical foundation, distributed systems- methods and tools for specification. *Springer-Verlag Lecture Notes in Computer Science*, 190, 1985.
7. K. Larsen. Ideal specification formalisms = expressivity + compositionality + decidability + testability + ... In *Invited Lecture at CONCUR 1990, LNCS 458*, 1990.
8. K. Larsen. The expressive power of implicit specifications. In *ICALP 1991, LNCS 510*, 1991.
9. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symp. Princ. of Prog. Lang.*, pages 97–107, 1985.
10. T. Margaria, H. Raffelt, and B. Steffen. Knowledge-based relevance filtering for efficient system-level test-based model generation (to appear). *Innovations in Systems and Software Engineering, a NASA Journal, Springer Verlag.*
11. T. Margaria, A. Sistla, B. Steffen, and L. D. Zuck. Taming interface specifications. In *CONCUR2005*, pages 548–561, 2005.
12. T. Margaria, A. Sistla, B. Steffen, and L. D. Zuck. Taming interface specifications. In *www. cs. uic. edu/~sistla*, 2005.
13. A. Pnueli, A. Zaks, and L. D. Zuck. Monitoring interfaces for faults. In *Proceedings of the 5th Workshop on Runtime Verification (RV'05)*, 2005. To appear in a special issue of ENTCS.
14. S. Safra. On the complexity of ω-automata. In *29th annual Symposium on Foundations of Computer Science, October 24–26, 1988, White Plains, New York*, pages 319–327. IEEE Computer Society Press, 1988.
15. A. Sistla, M. Zhou, and L. D. Zuck. Monitoring off-the-shelf components. A companion to the VMCAI06 paper, `www.cs.uic.edu/~lenore/pubs`, 2005.
16. A. P. Sistla. On characterization of safety and liveness properties in temporal logic. In *Proceedings of the ACM Symposium on Principle of Distributed Computing*, 1985.
17. A. P. Sistla. Proving correctness with respect to nondeterministic safety specifications. *Information Processing Letters*, 39:45–49, 1991.
18. M. Vardi, P. Wolper, and A. P. Sistla. Reasoning about infinite computations. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, 1983.

Parallel External Directed Model Checking with Linear I/O

Shahid Jabbar and Stefan Edelkamp

Computer Science Department, University of Dortmund,
Dortmund, Germany
{shahid.jabbar, stefan.edelkamp}@cs.uni-dortmund.de

Abstract. In this paper we present Parallel External A*, a parallel variant of external memory directed model checking. As a model scales up, its successors generation becomes complex and, in turn, starts to impact the running time of the model checker. Probings of our external memory model checker IO-HSF-SPIN revealed that in some of the cases about 70% of the whole running time was consumed in the internal processing. Employing a multiprocessor machine or a cluster of workstations, we can distribute the internal working load of the algorithm on multiple processors.

Moreover, assuming a sufficient number of processors and number of open file pointers per process, the I/O complexity is reduced to linear by exploiting a hash-function based state space partition scheme.

1 Introduction

In explicit-state model checking software [3], state descriptors are often so large, so that main memory is often not sufficient for a lossless storage of the set of reachable states during the exploration even if all available reduction techniques, like *symmetry* or *partial-order reduction* [20, 24] have been applied. Besides advanced implicit storage structures for the set of states [19] three different options have been proposed to overcome the internal space limitations for this so-called *state explosion* problem, namely, *directed*, *external* and *parallel* search.

Directed or heuristic search [23] guides the search process into the direction of the goal states, which in model checking safety properties is the set of software errors. The main observation is that using this guidance, the number of explored states needed to establish an error is smaller than with blind search. Moreover, *directed model checking* [29, 7] often reduces the length of the counter-example, which in turn eases the interpretation of the bug.

External search algorithms [26] store and explore the state space via hard disk access. States are flushed to and retrieved from disk. As virtual memory already can exceed main memory capacity, it can result in a slow-down of speed due to excessive page-faults if the algorithm lacks locality. Hence, the major challenge in a good design for an external algorithm is to control the locality of the file access, where block-transfers are in favor to random accesses. Since hashing has a bad reputation for preserving locality, in *external model checking* [27, 11] duplicate elimination is delayed by applying a subsequent external sorting and scanning

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 237–251, 2006.
© Springer-Verlag Berlin Heidelberg 2006

phase of the state set to be refined. During the algorithm only a part of the graph can be processed at a time; the remainder is stored on a disk. However, hard disk operations are about a $10^5 - 10^6$ times slower than main memory accesses. More severely, this latency gap rises dramatically. According to recent estimates, technological progress yields about annual rates of 40-60 percent increase in processor speeds, while disk transfers only improve by 7 to 10%. Moreover, the costs for large amount of disk space has considerably decreased. At the time of writing, 500 gigabytes can be obtained at the cost of 300-400 US dollars.

Parallel or distributed search algorithms[1] are designed to solve algorithmic problems by using many processors / computers. An efficient solution can only be obtained, if the organization between the different tasks can be optimized and distributed in a way that the working power is effectively used. *Distributed model checking* [28] tackles with the state explosion problem by profiting from the amount of resources provided by parallel environments. A speedup is expected if the load is distributed uniformly with a low inter-processes communication cost.

In *large-scale parallel breadth-first search* [14], the state space is fully enumerated for increasing depth. Using this approach a complete exploration for the *Fifteen-Puzzle* with 16!/2 states has been executed on six disks using a maximum of 1.4 terabytes of disk storage. In model checking such an algorithm is important to verify safety properties in large state spaces. In [11], the authors presented an external memory directed model checker that utilizes hard disk to store the explored states. It utilizes heuristics estimates to guide the search towards the error state.

In LTL model checking, as models scale up, the density of edges in the combined state space also increases. This in turn, effects the complexity of successors generation. Probing our external directed model checker IO-HSF-SPIN [11] revealed some bottlenecks in its running time. Surprisingly, in a disk-based model checker internal processing such as successor generations and state comparisons were sometimes dominating even the disk access times.

In this paper we present a parallel variant of external memory directed model checking algorithm that improves on our earlier algorithm in two ways. Firstly, the internal workload is divided among different processors that can either be residing on the same machine or on different machines. Secondly, we suggest an improved parallel duplicate detection scheme based on multiple processors and multiple hard disks. We show that under some realistic assumptions, we achieve a number of I/Os that is linear to the explored size of the model.

The paper is structured as follows. First we review *large-scale parallel breadth-first search*, the combined approach of external and parallel breadth-first search. Then, we turn to directed model checking and the heuristics used in model

[1] As it refers to related work, even for this text terminology is not consistent. In AI literature, the term *parallel search* is preferred, while in model checking research, the term *distributed search* is commonly chosen. In theory, parallel algorithms commonly refers to a synchronous scenario (mostly according a fixed architecture), while *distributed algorithms* are preferably used in an asynchronous setting. In this sense, the paper considers the less restricted distributed scenario.

checking. Next, we recall *External A**, the external version of the A* algorithm that serves as a basis to include heuristics to the search. Afterwards, we propose *Parallel External A** and provide algorithmic details for large-scale parallel A* search. Subsequently, we discuss some of the complexity aspects of the algorithm. Its application to distributed and external model checking domains is considered in the experimental part, where we have successfully extended the state-of-the-art model checker SPIN to include directed, external and parallel search. The paper closes with further links to related work.

2 Large-Scale Parallel Breadth-First Search

In *large-scale parallel breadth-first search* [14], the entire search space is partitioned into different files. The hash address is used to distribute and to locate states in those files. As the considered state spaces like the Fifteen-Puzzle are regular permutation games, each state can be perfectly hashed to a unique index. Since all state spaces are undirected, in order to avoid regenerating explored states, *frontier search* [15] stores, with each node, its used operators in form of a bit-vector in the size of the operator labels available. This allows to distinguish neighboring states that have already been explored from those that have not, and, in turn to omit the list of already explored states.

Hash-based delayed duplicate detection uses two orthogonal hash functions. When a state is explored, its children are written to a particular file based on the first hash value. In cases like the sliding-tile puzzle, the filename correspond to parts of state vector. For space efficiency it is favorable to perfectly hash the rest of the state vector to obtain a compressed representation. The representation as a permutation index can be computed in linear time w.r.t. the length of the vector.

Figure 1 depicts the layered exploration on the external partition of the state space. Even on a single processor, multi-threading is important to maximize the performance of disk-based algorithms. The reason is that a single-threaded implementation will run until it has to read from or write to disk. At that point it will block until the I/O operation has completed. Moreover hash-based delayed duplicate detection is well-suited to be distributed. Within an iteration, most file expansions and merges can be performed independently.

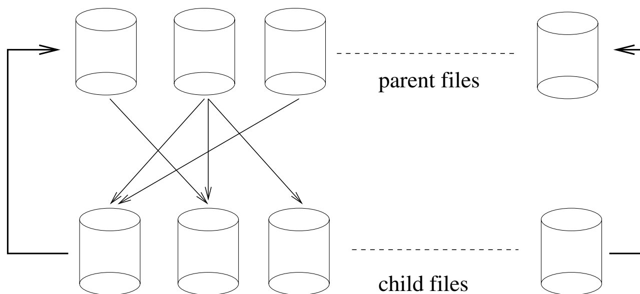


Fig. 1. Externally stored state space with parent and child files

To realize parallel processing a work queue is maintained, which contains parent files waiting to be expanded, and child files waiting to be merged. At the start of each iteration, the queue is initialized to contain all parent files. Once all the neighbors of a child file are finalized, it is inserted in the queue for merging. Each thread works as follows. It first locks the work queue. Two parent files conflict if they can generate states that hash to the same child file. The algorithm checks whether the first parent file conflicts with any other file currently being expanded. If so, it scans the queue for a parent file with no conflicts. It swaps the position of that file with the one at the head of the queue, grabs the non-conflicting file, unlocks the queue, and expands the file. For each child file it generates, it checks to see if all of its parents have been expanded. If so, it puts the child file to the queue for merging, and then returns to the queue for more work. If there is no more work in the queue, any idle threads wait for the current iteration to complete. At the end of each iteration the work queue is initialized to contain all parent files for the next iteration.

3 Directed Model Checking

Directed model checking [7] incorporates heuristic search algorithms like A* [23] to enhance the bug-finding capability of model checkers, by accelerating the search for errors and finding (near to) minimal counterexamples. In that manner we can mitigate the state explosion problem and the long counterexamples provided by some algorithms like depth-first search, which is often applied in explicit state model checking.

One can distinguish different classes of evaluation functions based on the information they try to exploit. *Property specific* heuristics [7] analyze the error description as the negation of the correctness specification. In some cases the underlying methods are only applicable to special kinds of errors. A heuristic that prioritizes transitions that block a higher number of processes focuses on deadlock detection. In other cases the approaches are applicable to a wider range of errors. For instance, there are heuristics for invariant checking that extract information from the invariant specification and heuristics that base on already given errors states. The second class has been denoted as being *structural* [9], in the sense that source code metrics govern the search. This class includes coverage metrics (such as *branch count*) as well as concurrency measures (such as *thread preference* and *thread interleaving*). Next there is the class of *user heuristics* that inherit guidance from the system designer in form of source annotations, yielding preference and pruning rules for the model checker.

4 External A*

*External A** [6] maintains the search horizon on disk. The priority queue data structure is represented as a list of buckets. In the course of the algorithm (cf. Figure 2), each bucket (i, j) will contain all states u with path length $g(u) = i$ and heuristic estimate $h(u) = j$. As similar states have same heuristic estimates,

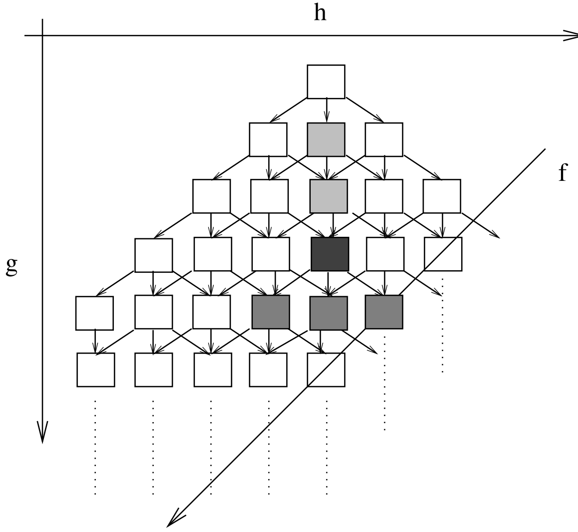


Fig. 2. Exploration in External A*

it is easy to restrict duplicate detection to buckets of the same h-value. By an assumed undirected state space problem graph structure, we can restrict aspirants for duplicate detection furthermore. If all duplicates of a state with g-value i are removed with respect to the levels i, $i-1$ and $i-2$, then no duplicate state will remain for the entire search process. For breadth-first-search in explicit graphs, this is in fact the algorithm of [22]. We consider each bucket as a different file that has an individual internal buffer. A bucket is *active* if some of its states are currently expanded or generated. If a buffer becomes full, then it is flushed to disk. The algorithm maintains the two values $g_{\min}$ and $f_{\min}$ to address the correct buckets. The buckets of $f_{\min}$ are traversed for increasing $g_{\min}$-value unless the $g_{\min}$ exceeds $f_{\min}$. Due to the increase of the $g_{\min}$-value in the $f_{\min}$ bucket, a bucket is finalized when all its successors have been generated. Given $f_{\min}$ and $g_{\min}$, the corresponding h-value is determined by $h_{\max} = f_{\min} - g_{\min}$. According to their different h-values, successors are arranged into different horizon lists. Duplicate elimination is delayed.

Since External A* simulates A* and changes only the order of elements to be expanded that have the same f-value, completeness and optimality are inherited from the properties of A*. The I/O complexity for External A* in an implicit unweighted and undirected graph with monotone heuristic is bounded by $O(sort(|E|) + scan(|V|))$, where $|V|$ and $|E|$ are the number of nodes and edges in the explored subgraph of the state space problem graph, and $scan(n)$ ($sort(n)$) are the number of I/Os needed to externally scan (sort) n elements.

For challenging exploration problems, external algorithms operate in terms of days and weeks. For improving fault-tolerance, we have added a *stop-and-resume* option on top of the algorithm, which allows the continuation of the exploration in case of a user interrupt. An interrupt causes all open buffers to be flushed and

the exploration can continue with the active buffer at the additionally stored file pointer position. In this case, we only have to redo at most one state expansion. As duplicates are eliminated, generating a successor twice does not harm the correctness and optimality of the algorithm. As we flush all open buffers when a bucket is finished, in case of severe failures e.g. to the power supply of the computer we have to re-run the exploration for at most one bucket.

I/O Efficient Directed Model Checking [11] applies variants of External A* to the validation of communication protocols. The tool IO-HSF-SPIN accepts large fractions of the Promela input language of the SPIN model checker. The paper extends External A* to weighted, directed graphs, with non-monotone cost functions as apparent in explicit state model checking and studies the scope for delayed duplicate within protocol verification domains.

5 Parallel External A*

The distributed version of External A* *Parallel External A** is based on the observation that the internal work in each individual bucket can be parallelized among different processors. Due to the dynamic allocation of new objects in software model checking, our approach is also compatible with state vectors of varying length. We first discuss our method of disk-based queues to distribute the work load among different processes. Our approach is applicable to both a client-server based environment or a single machine with multiple processors.

5.1 Disk-Based Queues

To organize the communication between the processors a working queue is maintained on disk. The working queue contains the requests for exploring parts of a (g, h) bucket together with the part of the file that has to be considered[2]. For improving the efficiency, we assume a distributed environment with one master and several slave processes[3]. Our approach applies to both the cases when each slave has its own hard disk or if they work together on one hard disk residing on the master. Message passing between the master and slave processes is purely done on files, so that all processes can run independently. For our algorithm, master and slave work fully autonomously. We do not use spawning of child processes. Even if slave processes are killed, their work can be re-done by any other idle process that is available.

One file that we call the *expand-queue*, contains all current requests for exploring a state set that is contained in a file. The filename consists of the current

[2] As processors may have different computational power and processes can dynamically join and leave the exploration, the number of state space parts under consideration do not necessarily have to match the number of processors. By utilizing a queue, one also may expect a processor to access a bucket multiple times. However, for the ease of a first understanding, it is simpler to assume that the jobs are distributed uniformly among the processors.

[3] In our current implementation the *master* is in fact an ordinary process defined as the one that finalized the work for a bucket.

g and h value. In case of larger files, file-pointers for processing parts of a file are provided, to allow for better load balancing. There are different strategies to split a file into equi-distance parts or into chunks depending on the number and performance of logged-on slaves. As we want to keep the exploration process distributed, we select the file pointer windows into equidistant parts of a fixed number of C bytes for the states to be expanded. For improved I/O, number C is supposed to divide the system's block size B. As concurrent read operations are allowed for most operating systems, multiple processes reading the same file impose no concurrency conflicts.

The expand-queue is generated by the master process and is initialized with the first block to be expanded. Additionally we maintain a total count on the number of requests, i.e., the size of the queue, and the current count of satisfied requests. Any logged-on slave reads a requests and increases the count once it finishes. During the expansion process, in a subdirectory indexed by the slave's name it generates different files that are indexed by the g and h value of the successor states.

The other queue is the *refine-queue* also generated by the master process once all processes are done. It is organized in a similar fashion as the expand queue and allows slaves to request work. The refine-queue contains filenames that have been generated above, namely the slave-name (that does not have to match with the one of the current process), the block number, and the g and h value. For a suitable processing the master process will move the files from subdirectories indexed by the slave's name to ones that are indexed by the block number. As this is a sequential operation executed by the master thread, we require that changing the file locations is fast in practice, an assumption that is fulfilled in all modern operating systems.

In order to avoid redundant work, each processor eliminates the requests from the queue. Moreover, after finishing the job, it will write an acknowledge to an associated file, so that each process can access the current status of the exploration, and determine if a bucket has been completely explored or sorted.

Since all communication between different processes is done through shared files, proper mechanism for mutual exclusion is necessary. We utilized a rather simple but efficient method to avoid concurrent writes accesses to the files. When ever a process has to write on a shared file, e.g., to the *expand-queue* to deque the request, it issues an operating system move (`mv`) command to rename the file into *<process ID>.expand-queue*, where *process ID* is a unique number that is automatically assigned to every process that enters the pool. If the command fails, it implies that the file is currently being used by another process. Since the granularity of a kernel-level command is much finer than any other program implemented on top of it, the above technique performed remarkably well.

5.2 Sorting and Merging

For each bucket that is under consideration, we establish four stages in the algorithm. These phases are visualized in Figure 3 (top to bottom). The zig-zag curves visualize the sorting order of the states sequentially stored in the files.

The sorting criteria is defined by the state's hash key, which dominates low-level state comparison based on the compressed state descriptor.

In the *exploration stage*, each processor p flushes the successors with a particular g and h value to its own file (g, h, p). Each process has its own hash table and eliminates some duplicates already in main memory. The hash table is based on chaining, with chains sorted along the state comparison function. However, if the output buffer exceeds memory capacity it writes the entire hash table to disk. By the use of the sorting criteria as given above, this can be done using a mere scan of the hash table.

In the *first sorting stage*, each processor sorts its own file. In a serial setting, such sorting has a limited effect on I/O when using external merge-sort afterwards. In a distributed setting, however, we exploit the advantage that the files can be sorted in parallel. Moreover, the number of file pointers needed is restricted by the number of flushed buffers, which is illustrated by the number of peaks in the figure. Based on this restriction, we only need to perform a merge of different sorted buffers - an operation in linear I/O.

In the *distribution stage*, a single processor distributes all states in the pre-sorted files into different files according to the hash value's range. This is a parallel scan with a number of file pointers that is equivalent to the number of files that have been generated. As all input files are pre-sorted this is a mere scan. No all-including file is generated, keeping the individual file sizes small. This is of course a bottleneck to the parallel execution, as all processes have to wait until the distribution stage is completed. However, if we expect the files to be on different hard drives, traffic for file copying is needed anyway.

In the *second sorting stage*, processors sort the files with buffers pre-sorted w.r.t the hash value's range, to find further duplicates. The number of peaks in each individual file is limited by the number of input files (= number of processors), and the number of output files is determined by the selected partitioning of the hash index range. The output of this phase are sorted and partitioned buffers. Using the hash index as the sorting key we establish that the concatenation of files is in fact totally sorted.

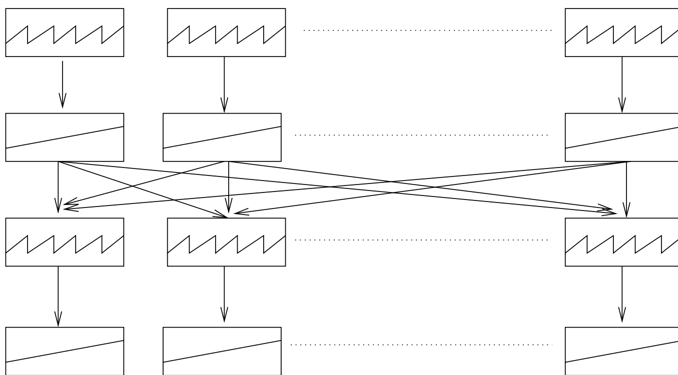


Fig. 3. Stages of bucket expansions in *Parallel External A**

6 Complexity

The complexity of external memory algorithm is usually measured in terms of I/Os, which assumes an unlimited amount of disk space. Using the complexity measurement [6] shows that in general it is not possible to exceed the external sorting barrier i.e., delayed duplicate detection. The lower bound for the I/O complexity for delayed duplicate bucket elimination in an implicit unweighted and undirected graph A* search with consistent estimates is at least $\Omega(sort(|E|))$, where E is the set of explored edges in the search graph. Fewer I/Os can only be expected if structural state properties can be exploited. We will see, that by assuming a sufficient number of processors and file pointers, the I/O complexity is reduced to linear (i.e. $\Omega(scan(|E|)) = |E|/|B|$ I/Os) by exploiting a hash-function based state space partition scheme. We assume that the hash function provides a uniform partitioning of the state space.

Recall that, the complexity in the external memory model assumes no restriction to the capacity of the external device. In practice, however, we have certain restrictions, e.g on the number of open file pointers per process.

We may, however, assume that the number of processes is smaller than the number of file pointers. Moreover, by the given main memory capacity, we can also assume that the number of flushed buffers (the number of peaks w.r.t. the sorted order of the file) is also smaller than the number of file pointers.

Using this we can achieve in fact a linear number of I/O for delayed duplicate elimination. The proof is rather simple. The number of peaks k in each individual file is bounded either by the number of flushed buffers or by the number of processes, so that a simple scan with k-file pointers suffices to finalize the sorting.

An important observation is that the more processors we invest, the finer the partitioning of the state space, and the smaller the individual file sizes in a partitioned representation. Therefore, a side effect on having more processors at hand is an improvement in I/O performance based on existing hardware resource bounds.

7 Experiments

We have implemented a slightly reduced version of *Parallel External A** in our extension to the model checker SPIN. Our tool, entitled IO-HSF-SPIN and first introduced in [11], is in fact an extension of the model checker HSF-SPIN developed by [18]. Instead of implementing all four stages for the bucket exploration, we restricted to three stages and use one server processor to merge the sorted outcome of the others. A drawback is that all processors have to wait for a finalized merge phase. Moreover, the size of the resulting file is not partitioned and, therefore, larger. Given that the time complexity is in fact hidden during the exploration of states, so far these aspects are not a severe limitation. As all input files are pre-sorted the stated I/O complexity of the algorithm is still linear.

We chose two characteristics protocols for our experiments, the CORBA-GIOP protocol as introduced by [12], and the Optical Telegraph protocol that comes with SPIN distribution. The CORBA-GIOP can be scaled according to two different parameters, the number of servers and the number of clients. We

selected three settings: *a*) 3 clients and 2 servers, *b*) 4 clients and 1 server, and *c*) 4 clients and 2 servers. For the Optical Telegraph, we chose one instance with 9 stations. CORBA-GIOP protocol has a longer state vector that puts much load on the I/O. On the other hand, the Optical Telegraph has a smaller state vector but takes a longer time to be computed which puts more load on the internal processing. Moreover, the number of duplicates generated in Optical Telegraph is much more than in CORBA-GIOP.

To evaluate the potential of the algorithm, we tested it on two different (rather small) infrastructures. In the first setting we used two 1 GHz Sun Solaris Workstations equipped with 512 megabyte RAM and a NFS mounted hard disk space. For the second setting we chose a Sun Enterprise System with four 750 MHz processors working with 8 gigabyte RAM and 30 gigabyte shared hard disk space. In both cases, we worked with a single hard disk, so that no form of disk parallelism was exploited. Throughout our experiments the sizes of individual processes remained less than 5% of the total space requirement.

Moreover, we used the system *time* command to calculate the CPU running times. The compiler used is GCC v2.95.3 with default optimizations and `-g` and `-pg` options turned on for debugging and profiling information. In all of the test cases, we searched for the deadlock using number of active processes as the heuristic estimate. We depict all three parameters provided by the system: *real* (the total elapsed time), *user* (total number of CPU-seconds that the process spent in user mode) and *system* (total number of CPU-seconds that the process spent in kernel mode). The speedup in the columns is calculated by dividing the time taken by a serial execution by the time taken by the parallel execution.

In the first set of experiments, the multi-processor machine is used. Table 1 and 2 depict the times[4] for three different runs consisting of single process, 2 processes, and 3 processes. The space requirements by a run of our algorithm is approx. 2.1 GB, 5.2 GB, 21 GB, and 4.3 GB for GIOP 3-2, 4-1, 4-2, and Optical-9, respectively. For GIOP 4-1, we see a gain by a factor of 1.75 for two processors and 2.12 for three processors in the total elapsed time. For Optical Telegraph, this gain went up to 2.41, which was expected due to its complex internal processing.

In actual CPU-time (*user*), we see an almost linear speedup that depicts the uniform distribution of internal workload and, hence, highlighting the potential of the presented approach.

Tables 3 and 4 show our results in the scenario of two machines connected together via NFS. In GIOP 3-2, we observe a small speed-up of a factor of 1.08. In GIOP 4-1, this gain increased to about a factor of 1.3. When tracing this limited gain, we found that the CPUs were not used at full speed. The bottleneck turned out to be the underlying NFS layer that was limiting the disk accesses to only about 5 Megabytes/sec. This bottleneck can be removed by utilizing local hard disk space for successors generation and then sending the files to the file server using secure copy (scp) that allows a transfer rate of 50 Megabytes/sec.

[4] The smallest given CPU time always corresponds to the process that established the error in the protocol first.

Table 1. CPU time for Parallel External A* in GIOP on a multiprocessor machine

GIOP 3-2	1 process	2 processes		Speedup	3 processes			Speedup
real	25m 59s	17m 30s	17m 29s	1.48	15m 55s	16m 6s	15m 58s	1.64
user	18m 20s	9m 49s	9m 44s	1.89	7m 32s	7m 28s	7m 22s	2.44
system	4m 22s	4m 19s	4m 24s	0.98	4m 45s	4m 37s	4m 55s	0.92
GIOP 4-1	**1 process**	**2 processes**		**Speedup**	**3 processes**			**Speedup**
real	73m 10s	41m 42s	41m 38s	1.75	37m 24s	34m 27s	37m 20s	2.12
user	52m 50s	25m 56s	25m 49s	2.04	18m 8s	18m 11s	18m 20s	2.91
system	10m 20s	9m 6s	9m 15s	1.12	9m 22s	9m 8s	9m 0s	1.13
GIOP 4-2	**1 process**	**2 processes**		**Speedup**	**3 processes**			**Speedup**
real	269m 9s	165m 25s	165m 25s	1.62	151m 6s	151m 3s	151m 5s	1.78
user	186m 12s	91m 10s	90m 32s	2.04	63m 12s	63m 35s	63m 59s	2.93
system	37m 21s	29m 44s	30m 30s	1.25	30m 19s	30m 14s	29m 50s	1.24

Table 2. CPU time for Parallel External A* in Optical Telegraph on a multiprocessor machine

Optical-9	1 process	2 processes		Speedup	3 processes			Speedup
real	55m 53s	31m 43s	31m 36s	1.76	23m 32s	23m 17s	23m 10s	2.41
user	43m 26s	22m 46s	22m 58s	1.89	15m 20s	14m 24s	14m 25s	3.01
system	5m 47s	4m 43s	4m 18s	1.34	3m 46s	4m 45s	4m 40s	1.22

Table 3. CPU time for Parallel External A* in GIOP on two computers and NFS

GIOP 3-2	1 process	2 processes		Speedup
real	35m 39s	32m 52s	33m 0s	1.08
user	11m 38s	6m 35s	6m 34s	1.76
system	3m 56s	4m 16s	4m 23s	0.91
GIOP 4-1	**1 process**	**2 processes**		**Speedup**
real	100m 27s	76m 38s	76m 39s	1.3
user	31m 6s	15m 52s	15m 31s	1.96
system	8m 59s	8m 30s	8m 36s	1.05

Table 4. CPU time for Parallel External A* in Optical Telegraph on two computers and NFS

Optical-9	1 process	2 processes		Speedup
real	76m 33s	54m 20s	54m 6s	1.41
user	26m 37s	14m 11s	14m 12s	1.87
system	4m 33s	3m 56s	3m 38s	1.26

In the Optical Telegraph, we see a bigger reduction of about a factor of 1.41 because of complex but small state vector and more dependency on internal computation. As in the former setting, the total CPU-seconds consumed by a process (*user*) in 1-process mode is reduced to almost half in 2-processes mode.

8 Related Work

There is much work on external search in explicit graphs that are fully specified with its adjacency list on disk. In model checking software the graph are implicit. There is no major difference in the exposition of the algorithm of Munagala and Ranade [22] for explicit and implicit graphs. However, the precomputation and access efforts are by far larger for the explicit graph representation. The breadth-first search algorithm has been improved by [21].

Even for implicit search, the body of literature is rising at a large pace. Edelkamp and Schrödl [8] consider external route-planning graphs that are naturally embedded into the plane. This yields a spatial partitioning that is exploited to trade state exploration count for improved local access. Zhou and Hansen [30] impose a projection function to have buckets to control the expansion process in best-first search. The projection preserves the duplicate scope or locality of the state space graph, so that states that are outside the locality scope do not need to be stored. Korf [13] highlights different options to combine A*, frontier and external search. His proposal is limited as only any two options were compatible. Edelkamp [5] extends the External A* with BDDs to perform a external symbolic BFS in abstract space, followed by an external symbolic A* search in original space that take the former result as a lower bound to guide the search. Zhou and Hansen [31] propose structure preserving state space projections to have a reduced state space to be controlled on disk. They also propose external construction of pattern databases. The drawback of their approach is that it applies only to the state spaces that have a very regular structure - something that is not available in model checking.

In Stern and Dill's initial paper on external model checking in the Murϕ Verifier variants of external breadth-first search are considered. In Bao and Jones [1], we see another faster variant of Murϕ Verifier with magnetic disk. They propose two techniques: one is based on partitioned hash tables, and the other on chained hash table. They targeted to reduce the delayed duplicate detection time by partitioning the state space that, in turn, diminishes the size of the set to be checked. They claim their technique to be inherently serial having less room for a distributed variant. In the approach of Kristensen and Mailund [16] repeated scans over the search space in a geometric scan-line approach with states that are arranged in the plane wrt. some *progress measure* based on a given partial order. The scan over the entire state space is memory efficient, as it only needs to store the states that participate in the transitions that cross the current scan position. These states are marked visited and the scan over the entire state space is repeated to cope with states that are not reachable with respect to earlier scans. Their dependeny on a good *progress measure* hinders its applicability to model checking in general. They have applied it mainly to Petri nets based model checking where the notion of *time* is used as a *progress measure*.

While some approaches to parallel and distributed model checking are limited to the verification of safety properties [2, 10, 17], other work propose methods for checking liveness properties expressed in linear temporal logic (LTL) [4, 18]. Recall that LTL model checking mainly entails finding accepting cycles in a

state space, which is performed with the nested depth-first search algorithm. The correctness of this algorithm depends on the depth-first traversal of the state space. Since depth-first search is inherently sequential [25], additional data structures and synchronization mechanisms have to be added to the algorithm. These requirements can waste the resources offered by the distributed environment. Moreover, formally proving the correctness of the resulting algorithms is not easy. It is possible, however, to avoid these problems by using partition functions that localize cycles within equivalence classes. The above described methods for defining partitions can used for this purpose, leading to a distributed algorithm that performs the second search in parallel. The main limitation factor is that scalability and load distribution depend on the structure of the model and the specification.

Brim et. al. [4] discusses one such approach where the SPIN model checker has been extended to perform nested depth-first search in a distributed manner. They proposed to maintain a dependency structure for all the accepting states visited. The nested parts for these accepting states are then started as separate procedures based on the order dictated by the dependeny structure. Lluch-Laufente [18] improves on the idea of nested depth-first search. The proposed idea is to divide the state space in strongly connected components by exploiting the structure of the *never claim* automaton of the specification property. The nested search is then restricted only to the corresponding component. If during exploration, a node that belongs to some other component is encountered, it is inserted in the *visited* list of its corresponding component. Unfortunately, there is little or almost no room externalize the above two approaches. Depth-first search lacks *locality* and hence not suited to be externalize. Stern and Dill [28] propose a parallel version of the Murϕ model checker. They also use a scheme based on run-time partitioning of the state space and assigning different partitions to different processors. The partitioning is done by using a universal hash function that uniformly distributes the newly generated states.

9 Conclusion

Enhancing directed model checking is essential to improve error detection in software. The paper contributes the first study of combining external, directed and parallel search to mitigate the state-explosion problem in model checking. We have shown a successful approach to extend the external A* exploration in a distributed environment, as apparent in multi-processor machines and workstation clusters. Exploration and delayed duplicate detection are parallelized without concurrent write access, which is often not available.

Error trails provided by depth-first search exploration engines are often exceedingly lengthy. Employed with a lower-bound heuristic, the proposed algorithm yields counter-examples of optimal length, and is, therefore, an important step to ease error comprehension for the programmer / software designer. Under reasonable assumptions on the number of file pointers per process, the number of I/Os is linear in the size of the model, by means the external work of exploring the model matches the complexity of scanning it.

The approach is implemented on top of the model checker IO-HSF-SPIN and the savings for the single disk model are very encouraging. We see an almost linear speedup in the CPU-time and significant gain in the total elapsed time. Compared to the potential of external search, the models that we have looked at are considerably small. In near future, we expect to implement the multiple-disk version of the algorithm as mentioned in this text. To conduct empirical observations for external exploration algorithm is a time-consuming task. In very large state spaces algorithms can run for weeks. For example the complete exploration of the Fifteen Puzzle consumed more than three weeks. Given more time, we expect larger models to be analyzed. We also expect further fine-tuning to increase the speed-up that we have obtained. Moreover, the approach presented is particular to model checking only, and can be applied to other areas where searching in a large state space is required.

Acknowledgments. The authors wish to thank Mathias Weiss for helpful discussions and technical support that made running of the presented experiments possible.

References

1. T. Bao and M. Jones. Time-efficient model checking with magnetic disks. In *Tools and Algorithms for the Construction and Analysis of Systems(TACAS)*, pages 526–540, 2005.
2. S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal methods in Computer-Aided Design (FMCAD)*, pages 390–404, 2000.
3. B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification.* Springer, 2001.
4. L. Brim, J. Barnat, and J.Stribnra. Distributed LTL model-checking in SPIN. In *Workshop on Software Model Checking (SPIN)*, pages 200–216, 2001.
5. S. Edelkamp. External symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 51–60, 2005.
6. S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *German Conference on Artificial Intelligence(KI)*, volume 3238, pages 226–240, 2004.
7. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2–3):247–267, 2004.
8. S. Edelkamp and S. Schrödl. Localizing A*. In *National Conference on Artificial Intelligence (AAAI)*, pages 885–890, 2000.
9. A. Groce and W. Visser. Heuristic model checking for java programs. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.
10. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *International Conference on Computer-Aided Verification (CAV)*, pages 20–35, 2000.
11. S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 3385, pages 313–329, 2005.

12. M. Kamel and S. Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, 2000.
13. R. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 650–657, 2004.
14. R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1380–1385, 2005.
15. R. E. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence allignment. In *National Conference on Artificial Intelligence (AAAI)*, pages 910–916, 2000.
16. L. M. Kristensen and T. Mailund. Path finding with the sweep-line method using external storage. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 319–337, 2003.
17. F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Workshop on Software Model Checking (SPIN)*, 1999.
18. A. Lluch-Lafuente. *Directed Search for the Verification of Communication Protocols*. PhD thesis, University of Freiburg, 2003.
19. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.
20. K. L. McMillan. Symmetry and model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 117–137. Springer-Verlag, 1998.
21. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, pages 723–735, 2002.
22. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 87–88, 2001.
23. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
24. D. A. Peled. Ten years of partial order reduction. In *Computer-Aided Verification (CAV)*, volume 1427, pages 17–28, 1998.
25. J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.
26. P. Sanders, U. Meyer, and J. F. Sibeyn. *Algorithms for Memory Hierarchies*. Springer, 2002.
27. U. Stern and D. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *International Conference on Computer Aided Verification (CAV)*, pages 172–183, 1998.
28. U. Stern and D. L. Dill. Parallelizing the Murphi verifier. In *International Conference on Computer-Aided Verification (CAV)*, pages 256–278, 1997.
29. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.
30. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, 2004. 683–689.
31. R. Zhou and E. Hansen. External-memory pattern databases using delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 1398–1405, 2005.

Piecewise FIFO Channels Are Analyzable

Naghmeh Ghafari and Richard Trefler⋆

School of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada

Abstract. FIFO systems consisting of several components that communicate via unbounded perfect FIFO channels arise naturally in modeling distributed systems. Despite well-known difficulties in analyzing such systems, they are of significant interest as they can describe a wide range of Internet-based communication protocols. Previous work has shown that the piecewise languages play important roles in the study of FIFO systems. In this paper, we show that FIFO systems composed of piecewise components can in fact be analyzed algorithmically. We demonstrate that any FIFO system composed of piecewise components can be described by a finite state, *abridged* structure, representing an expressive abstraction of the system. We present a procedure for building the abridged model and prove that this procedure terminates. We show that we can analyze the infinite computations of the more concrete model by analyzing the computations of the finite, abridged model. This enables us to check properties of the FIFO systems including safety properties of the components as well as a general class of end-to-end system properties. Finally, we apply our analysis method to an IP-telecommunication architecture to demonstrate the utility of our approach.

1 Introduction

Finite state machines that communicate over unbounded channels are used as a model of computation in the analysis of distributed protocols (*cf.* for example [10, 6, 1, 19, 12]). While unboundedness of communication channels simplifies the modeling of the protocols, it complicates their analysis. Since one unbounded channel is sufficient to simulate the tape of a Turing machine, most interesting verification problems for this class of protocols are undecidable. However, a substantial effort has gone into identifying subclasses for which the verification problem is decidable because this analysis is crucial in the design of safety-critical distributed systems (*cf.* [1, 2, 4, 5, 6, 8, 9, 12, 16, 17, 19]).

In this paper, we show that by restricting attention to systems composed of a class of 'well-designed' components, automated system analysis is possible even when the components communicate over unbounded perfect FIFO channels. This work was inspired by studying real world examples of distributed protocols, such as IP-telecommunication protocols [7]. In those protocols, it is particularly desirable that communication between peers (or components) be well-behaved.

⋆ Authors are supported in part by grants from OGS, NSERC of Canada, and Nortel Networks.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 252–266, 2006.
© Springer-Verlag Berlin Heidelberg 2006

And in fact, many descriptions of components with well-behaved communication can be expressed by a subclass of regular languages known as the *piecewise* languages (*cf.* [17, 7, 15, 25]).

Piecewise regular languages play important roles in the study of communication protocols (*cf.* [9, 17]). Intuitively, a language is piecewise if it is the union of sets of strings, where each set is given by a regular expression of the form $M_0^* a_0 M_1^* \ldots M_{n-1}^* a_{n-1} M_n^*$, in which each M_i is a subset of an alphabet Σ and each a_i is an element of Σ. Surprisingly, these relatively straightforward expressions can be used to capture important system properties.

In this paper, we show that in fact FIFO systems composed of communicating piecewise components are amenable to algorithmic analysis. It was shown in [17] that the limit language of the channel contents of such systems is regular. A method was also presented for calculating the limit language of systems with only a single channel and a certain class of multiple channel systems. Yet, calculating the limit language of general multiple channel systems remains an open problem. However, the strings that represent the contents of the channels may be too long to be calculated exactly. Thus, a method of abridging channel contents (without losing key information) is required.

First, we present a procedure to calculate, for each channel, a superset of the channel language associated with specific component states. We then use these supersets in calculating representations of the system computations. We note that, in general, limit languages do not represent system computations but rather the reachable state sets. Further, we show that our procedure is applicable to all FIFO systems composed of piecewise components. It is worth mentioning that while a string in the superset may not ever occur in a reachable system state, for the analysis technique presented in the current work, reachability of component states is exact.

We present a procedure that translates an n process distributed state machine ($DSM(n)$) composed of piecewise components communicating over FIFO channels into an abridged distributed state machine ($ADSM(n)$). The $ADSM(n)$ is closely related to $DSM(n)$; however, it differs in that the contents of the unbounded channels are represented by piecewise regular expressions. We establish the finiteness of the abridged model by proving that the calculation procedure terminates. Furthermore, we show that a global, component state, composed of the local states of the processes, is reachable in $ADSM(n)$, if and only if, the same component state is reachable in the corresponding $DSM(n)$. The representation of channel contents by piecewise regular expressions in the context of global system transitions has allowed us to group together sets of actions that may be executed by one process from a given global state.

The main reason for abridging $DSM(n)$ is to be able to reason about its infinite behavior by analyzing the behavior of the finite $ADSM(n)$. We consider system properties expressed by a restricted, but expressive, class of Büchi automata. Here, the states of the Büchi automata represent the finite set of component states and we require that the language of the automata be stuttering closed. We can then show that there is a computation of $ADSM(n)$ that satisfies

the automaton, if and only if, there is a computation of $DSM(n)$ that satisfies the automaton. This procedure allows one to check properties of the $DSM(n)$ including both local and global reachability properties as well as a general class of end-to-end system properties, thus showing that piecewise FIFO channels are analyzable.

1.1 Motivation

IP-telecommunication protocols are often prone to subtle errors because of indeterminacy in the scheduling of concurrent processes and communication latency. These errors can easily go undetected in testing and simulation due to their infrequency yet they can cause major destruction when they occur. Thus, it is desirable to formally verify that the protocols meet their specifications in all circumstances (*cf.* [1, 2, 5, 7, 12, 14, 19]). IP-telecommunication protocols can in many cases be effectively modeled as finite state machines communicating via unbounded channels, with enough generality to examine the concurrency issues involved.

ECLIPSE, now called BoxOS, is the next generation telecommunication services over IP infrastructure developed at AT&T Labs (see [7, 15, 25] and [17] for related work). Essentially, a telephone call is represented by a list (or more generally, a graph) of boxes, while communication between neighboring boxes is handled by perfect FIFO channels. At a sufficient level of abstraction, boxes may all be viewed as finite state transducers. Importantly, the language of the automata that model the communication behavior of these boxes is naturally a piecewise regular language. Communication in these protocols begins with an initiator trying to reach a given destination. A call is built recursively. The current endpoint begins the call initiation protocol with a chosen neighbor, the callee. If this initiation results in a stable connection, the callee becomes the new endpoint and the call construction continues. Call termination is required to proceed in a reverse order and in general required to begin at a call endpoint.

Here, our focus is on an important class of properties that involve several boxes; for instance, that particular messages sent from a local box eventually reach a distant box. While communication with multiple neighbors significantly complicates the description of the box, the languages of box inputs, box outputs, and sequences of box states can each be given by piecewise expressions.

1.2 Previous Work

FIFO systems have played key roles in the description of distributed systems. A complete description of prior work is given in [17]. Brand and Zafiropulo [10] first showed that many important questions regarding FIFO channels are undecidable. Despite this, Pachl [19] described several scenarios that are tractable. In [10], Brand and Zafiropulo also defined a number of minimal properties that *well-formed* protocols are expected to satisfy and showed to what extent these properties can be ensured. The work of [3] addresses the problem of automatically validating protocols based on algorithms that can detect violations of such minimal properties.

FIFO nets, a generalization of FIFO systems, are described in a survey [14] along with several decidability results that depend on the use of bounded languages. The work of [13] considers the use of semilinear regular expressions to represent bounded languages. We note that piecewise languages are not required to be bounded.

Several important decidability results have been given in the context of lossy channel systems [1, 2, 12]. For lossy systems, messages may be lost arbitrarily. Simple regular expressions are considered in [1] in the study of lossy channels. In contrast, our model is not lossy; some messages may be duplicated but they may not be lost.

The recent work on regular model checking [8, 16] (see also [9]) has been focused on the analysis of parameterized systems, whereas in this paper, we have provided automated analysis techniques applicable to specific FIFO systems.

Surprisingly, Boigelot and Godefroid [5] were able to show BDD-based symbolic methods could be used to calculate the limit language of channel contents for many important programs. Further, [4] extends their work to consider sets of operations on channel contents, while [6] considers sequences of channel operations that preserve the regularity of channel languages. We have instead focused on the reachability of component states for a specific class of protocols that allow conditional write operations; it is these conditional operations that increase the expressiveness of our model.

Piecewise regular languages are strictly more expressive than the piecewise testable languages of [23]. Further, piecewise languages have been described as Alphabetic Pattern Constraints [9]. These languages have also been considered in [20] and [21]. However, these works did not consider use of piecewise languages in the analysis of FIFO systems.

2 Piecewise Languages

Let Σ be a finite alphabet, $\mathbb{N}$ denote the set of natural numbers, and λ the empty string. Given two strings r_1 and r_2, concatenation of the elements of r_1 and r_2 is denoted by $r_1 r_2$. If r^* represents the Kleene closure of r, then $r^+ = rr^*$. Let $a \in \Sigma$, $l \in \mathbb{N}$, for $i \in [0..l]$, $a_i \in \Sigma$. In the sequel, $a_1 + a_2$ denotes the nondeterministic choice between a_1 and a_2. By $\langle +a_i \mid \mathrm{P}(a_i) \rangle$, we denote the regular expression $(a_1 + \ldots + a_m)$ consisting of the a_i's that satisfy P.

Definition 1. *[17] Thin piecewise regular (tpr) expressions are defined by the following grammar* $r ::= \lambda \mid a \mid (a_1 + \ldots + a_l)^+ \mid r_1 r_2$.

For example, $(a + b)^+ac$ is a *tpr* expression but $(ab)^+c$ is not. We consider *tpr*'s of the form λ, a, and $(a_1 + \ldots + a_l)^+$ to be *atomic*.

Definition 2. *[17] Piecewise regular (pr) expressions generalize tpr expressions allowing the inclusion of the + operator.*

For example, given thin piecewise regular expressions r_1 and r_2, $r_1 + r_2$ is a *pr*. It should be noted that $(ab)^+c$ is not in fact piecewise.

Proposition 1. *(cf. [17, 9]) Piecewise languages are star-free; they are closed under finite unions, finite intersections, concatenation, shuffle, projections (defined by letter-to-letter mappings), and inverse homomorphisms, but not under complementation and substitutions.*

Definition 3. *(cf. [17]) A piecewise automaton* $A = ((\Sigma, Q, \delta, q^0, F), \leq)$ *is defined as follows:* Q *is a finite set of states;* $q^0 \in Q$ *is the initial state;* $\leq$ *is a partial order on* Q*;* $\delta : Q \times \Sigma \rightarrow 2^Q$ *is a non-deterministic transition relation and if* $q' \in \delta(q, a)$ *then* $q \leq q'$*;* $F \subseteq Q$ *is the set of accepting states. We sometimes omit* F*, in which case it is understood that* $F = Q$*.*

Given $q \in Q$ and $w \in \Sigma^*$, $\delta(q, w)$ is defined as usual: $\delta(q, \lambda) = \{q\}$ and $\delta(q, wa) = \{p \mid \text{for some state } r \in \delta(q, w), p \in \delta(r, a)\}$. For $w \in \Sigma^*$ we say that the piecewise automaton A accepts w iff $\delta(q^0, w)$ contains a state in F. The language of A is defined as $\mathcal{L}(A) = \{w \in \Sigma^* \mid \delta(q^0, w) \text{ contains a state in } F\}$.

Proposition 2. *[17] For every piecewise automaton* A*, there is a piecewise regular expression* r *such that* $\mathcal{L}(A) = r$*, and for every piecewise regular expression* r*, there is a piecewise automaton* A *such that* $\mathcal{L}(A) = r$*.*

3 FIFO Channel Systems

In this section, we define $DSM(n)$ and its abridged model, $ADSM(n)$. Then we present a procedure to construct $ADSM(n)$ from a given $DSM(n)$ description and we prove that this procedure terminates. We also show the relationship between the computations of $ADSM(n)$ and $DSM(n)$.

In the sequel, the A_i's are piecewise automata, $A_i = ((\Sigma_i, Q_i, \delta_i, q_i^0), \leq_i)$. These automata may read from a single incoming channel, write on a single outgoing channel, or conditionally, read from a single incoming channel and write on a single outgoing channel. A distributed state machine with n piecewise automata is an asynchronous system with $n^2 - n$ channels and is defined as follows.

Definition 4. *For a set of piecewise automata* $\{A_i\}_{i \in [0..n-1]}$*, the* $DSM(n) = (Q, C, \Sigma, R, q^0, \delta)$ *is given by:*

- $Q = \times_i Q_i$ *is the component state set.*
- $C = \{c_{0,1}, \ldots, c_{n-1,n-2}\}$ *is a set of channels,* $c_{i,j}$ *is the channel from process* i *to process* j*.*
- $\Sigma = \cup_{c \in C} \Sigma_c$*,* Σ_c *is the alphabet of channel* c*.*
- $R = \times_{c \in C} \Sigma_c^*$ *is the set of possible channel descriptions.*
- q^0 *is the initial state.*
- δ *is the transition relation:*

$$\delta \subseteq Q \times \left(\bigcup_{c, c \in C} \{c?a, c!b, c?a \rightarrow c'!d \mid a, b \in \Sigma_c \text{ and } d \in \Sigma_c \} \right) \times Q$$

and is given below.

Intuitively, the transition relation δ is built up from the transition relations of the piecewise A_i's such that every transition in δ consists of exactly one transition in δ_i. Then δ is a set of triples (q, op, q'), where q and q' are in Q and op is a read, write, or a conditional operation. Thus, a transition of the form $(q, c?a, q')$ represents a change of the component state q to q' while removing an a from the head of channel c. The channel content must be of the form aw for this operation to be enabled. A transition of the form $(q, c!b, q')$ represents a change of the component state q to q' while transforming the content of the channel c from w to wb. A transition of the form $(q, c?a \rightarrow c'!d, q')$ represents a change of the component state q to q' while removing a from the head of channel c and appending d to the tail of channel c'. Since every transition in δ consists of exactly one transition in δ_i, we may abuse the notation by using $(q_i, op, q_i') \in \delta$ instead of $(q, op, q') \in \delta$ where q_i and q_i' are in Q_i.

A global state of $DSM(n)$ is composed of two parts: component state, which presents the local states of the processes, and the contents of the channels.

Definition 5. *A global state of $DSM(n)$ is a tuple $\psi = (q_0, \ldots, q_{n-1}, r_{0,1}, \ldots, r_{n-1,n-2})$ with $q_i \in Q_i$ and $r_{i,j} \in \Sigma_{i,j}^*$. The initial global state is $q^0 = (q_0^0, \ldots, q_{n-1}^0, \lambda, \ldots, \lambda)$.*

We use the following notation to refer to the elements of a global state: $\psi(i) = q_i$ and $\psi(i,j) = r_{i,j}$. We assume that the alphabets of the different channels are pairwise disjoint. Thus, $\delta_i(q_i, a)$ is a read transition if $a \in \Sigma_{j,i}$ or a write transition if $a \in \Sigma_{i,j}$.

The *global transition relation* of $DSM(n) = (Q, C, \Sigma, R, q^0, \delta)$ is a set $\mathcal{G}$ of triples (ψ, op, ψ'), where ψ and ψ' are global states. We will write $\psi \xrightarrow{op} \psi'$ to denote that ψ' is a successor of ψ and $(\psi, op, \psi') \in \mathcal{G}$ is given below:

- if $(q_i, c_{k,i}?a, q_i') \in \delta$, then $\psi \xrightarrow{c_{k,i}?a} \psi'$ provided that $\psi(i) = q_i$, $\psi'(i) = q_i'$, $\psi(k,i) = a\psi'(k,i)$, for all $j \neq i$, $\psi'(j) = \psi(j)$, and for all $(l,m) \neq (k,i)$, $\psi'(l,m) = \psi(l,m)$.
- if $(q_i, c_{i,k}!b, q_i') \in \delta$, then $\psi \xrightarrow{c_{i,k}!b} \psi'$ provided that $\psi(i) = q_i$, $\psi'(i) = q_i'$, $\psi'(i,k) = \psi(i,k)b$, for all $j \neq i$, $\psi'(j) = \psi(j)$, and for all $(m,l) \neq (i,k)$, $\psi'(m,l) = \psi(m,l)$.
- if $(q_i, c_{k,i}?a \rightarrow c_{i,j}!d, q_i') \in \delta$, then $\psi \xrightarrow{c_{k,i}?a \rightarrow c_{i,j}!d} \psi'$ provided that $\psi(i) = q_i$, $\psi'(i) = q_i'$, $\psi(k,i) = a\psi'(k,i)$, $\psi'(i,j) = \psi(i,j)d$, for all $u \neq i$, $\psi'(u) = \psi(u)$, and for all $(l,m) \neq (k,i)$ and $(l,m) \neq (i,j)$, $\psi'(l,m) = \psi(l,m)$.

We write $\psi \rightarrow \psi'$ when we do not distinguish the specific operation that causes the change of global states from ψ to ψ'.

Definition 6. *A computation path in $DSM(n)$ is a finite or infinite sequence denoted $\psi = \psi_0 \rightarrow \psi_1 \rightarrow \ldots$ where $\psi_0 = q^0$ and for all i, $\psi_i \rightarrow \psi_{i+1} \in \mathcal{G}$.*

All the computation paths in $DSM(n)$ are acceptable. Then $\mathcal{L}^*(DSM(n))$ is a subset of $(Q \times R)^*$ and consists of all the finite computations in $DSM(n)$ and

$\mathcal{L}^\omega(DSM(n))$ is a subset of $(Q \times R)^\omega$ and consists of all the infinite computations in $DSM(n)$. The language of $DSM(n)$, denoted $\mathcal{L}(DSM(n))$, is equal to $\mathcal{L}^*(DSM(n)) \cup \mathcal{L}^\omega(DSM(n))$.

3.1 Construction of $ADSM(n)$

We give a definition for $ADSM(n)$ that contains a procedural definition of its transition relation. This procedural definition describes how to build $ADSM(n)$ in an automated way from the syntactic description of $DSM(n)$.

$ADSM(n)$ is closely related to $DSM(n)$. Its component state set is the same as the component state set of $DSM(n)$. However, the channel contents in $DSM(n)$ are replaced by *tpr* expressions in $ADSM(n)$.

Let $tpr(\Sigma_c)$ denote the set of *tpr*'s over the alphabet Σ_c. Given a $DSM(n)$, the $ADSM(n)$ is defined as follows.

Definition 7. *For a given* $DSM(n) = (Q, C, \Sigma, R, q^0, \delta)$*, the* $ADSM(n) = (Q, T, q^0, \eta, \Phi)$ *where*

- $Q = \times_i Q_i$ *is the component state set.*
- $T = \times_{c \in C}\ tpr(\Sigma_c)$ *is the set of possible channel descriptions.*
- q^0 *is the initial state.*
- $\eta \subseteq (Q \times T) \times (Q \times T)$ *is the transition relation and is given below.*
- Φ *denotes a fairness constraint on the transitions.*

The fairness constraint requires that a transition that only reads from a channel should not be allowed to read an empty channel: for all processes, if process i infinitely often reads a from channel $c_{j,i}$, then process j must infinitely often writes a on channel $c_{j,i}$.

Definition 8. *A global state of* $ADSM(n)$ *is a tuple* $\xi = (q_0, \ldots, q_{n-1}, t_{0,1}, \ldots, t_{n-1,n-2})$ *with* $q_i \in Q_i$ *and* $t_{i,j} \in tpr(\Sigma_{i,j})$*. The initial global state is* $q^0 = (q_0^0, \ldots, q_{n-1}^0, \lambda, \ldots, \lambda)$*.*

In order to motivate the definition of transition relation in $ADSM(n)$, we first explain how the contents of the channels are updated by a single transition. If there is a write self loop operation ($c!a$) on a state of a process, we cannot say in advance how many times this transition is executed. However, here the assumption is that the $c!a$ is executed at least once. Thus, we represent the set of all write transitions ($c!a$) in $DSM(n)$ by a single transition in $ADSM(n)$ that writes a^+ on the channel. Now, assume another process has a self loop read operation $c?a$. Again, we do not know in advance how many times this transition may be executed. Thus, the corresponding $ADSM(n)$ also accepts an infeasible computation in which there are more read operations than the number of a's in the channel. However, this does not cause any problems since $ADSM(n)$ also accepts a *similar* computation in which the number of read and write operations are equal.

Example: $ADSM(n)$ Transition Relation. Consider the ith process in $DSM(n)$, A_i. Assume A_i, on state s, has only one self loop transition with a conditional operation $c_{j,i}?a \to c_{i,k}!b$. Let $\xi = (\ldots, s_i, \ldots, t_{j,i}, t_{i,k}, \ldots)$ be a global state of the corresponding $ADSM(n)$. Let $\xi' = (\ldots, s'_i, \ldots, t'_{j,i}, t'_{i,k}, \ldots)$ be a possible next global state of ξ that corresponds to the self loop conditional

transition $c_{j,i}?a \rightarrow c_{i,k}!b$ on state s of A_i. It is clear that $s'_i = s_i$. The following shows how the contents of channel $c_{j,i}$ and $c_{i,k}$ are updated in ξ'.

channel $c_{i,k}$*:*

- If $t_{i,k} = wb^+$, then there will be no change in the content of $c_{i,k}$; thus $t'_{i,k} = t_{i,k}$.
- If $t_{i,k} \neq wb^+$, then b^+ is appended to the tail of $c_{i,k}$; thus $t'_{i,k} = t_{i,k}b^+$.

channel $c_{j,i}$*:*

- If $t_{j,i} = aw$, then a gets read and the content of $c_{j,i}$ transforms to w.
- If $t_{j,i} = (a + a_1 + \ldots + a_m)^+w$, then there are four possible values for $t'_{j,i}$:
  - The head of $t_{j,i}$ may represent a string that starts with a followed by another a. Thus, by reading one a the content of the channel $c_{j,i}$ still can be represented by $(a + a_1 + \ldots + a_m)^+w$. As a result $t'_{j,i} = t_{j,i}$.
  - The head of $t_{j,i}$ may represent a string with length one, a. Thus, by reading one a the content of the channel $c_{j,i}$ transforms to w.
  - The head of $t_{j,i}$ may represent a string that starts with a followed by a string that does not contain any a's. Thus, the content of $c_{j,i}$ transforms to $(a_1 + \ldots + a_m)^+w$.
  - The head of $t_{j,i}$ may represent a string that starts with a, followed by a string that consists of letters from the set $\{a_1, \ldots, a_m\}$, followed by a string that contains a's. Thus, the content of channel $c_{j,i}$ transforms to $(a_1 + \ldots + a_m)^+(a + a_1 + \ldots + a_m)^+w$.

End of Example.

The representation of channel contents by thin piecewise regular expressions in the context of global system transitions allows us to group together sets of transitions that may be executed by one process from a given global state.

For the sake of clarity, we present the read and write transitions in the format of conditional transitions. Thus, a write transition is presented as a conditional transition that reads a dummy message from any of the incoming channels. A read transition is presented as a conditional transition that writes a dummy message on an outgoing channel.

Let α_j denote the head of channel $c_{j,i}$. Let $\beta_{j,k} \subseteq \alpha_j$ be a set of letters at the head of channel $c_{j,i}$ that enables a set of self loop conditional transitions that writes on $c_{i,k}$ at state q of process A_i: $\beta_{j,k} = \{b_{ji} \in \Sigma_{j,i} \mid \text{there is an } e \in \Sigma_{i,k} \text{ and } q \in \delta_i(q, (b_{ji}, e))\}$. Let $\beta'_{j,k} \subseteq \alpha_j$ be a set of letters at the head of channel $c_{j,i}$ that enables a set of conditional transitions to other states that writes on $c_{i,k}$ at state q of A_i: $\beta'_{j,k} = \{b'_{ji} \in \Sigma_{j,i} \mid \text{there is an } e \in \Sigma_{i,k} \text{ and } q' \in \delta_i(q, (b'_{ji}, e)) \text{ and } q' \neq q\}$. Let $\beta_{i,k} = \cup_j \beta_{j,k}$, $\beta'_{i,k} = \cup_j \beta'_{j,k}$ and $\beta_{j,i} = \cup_k \beta_{j,k}$.

In the sequel, let $\{e_{ik}\}$ be a set of letters that may be written on $c_{i,k}$ due to a set of enabled self loop conditional transitions at state q of A_i and $\epsilon_k = \langle +e_{ik} \mid \text{for some } b \in \beta_{i,k} \text{ and } q \in \delta_i(q, (b, e_{ik}))\rangle$. Let $\{e'_{ik}\}$ be a set of letters that may be written on $c_{i,k}$ due to a set of enabled conditional transitions to other states at state q of A_i and $\epsilon'_k = \langle +e'_{ik} \mid \text{for some } b' \in \beta'_{i,k},\ q' \in \delta_i(q, (b', e'_{ik})) \text{ and } q' \neq q\rangle$.

The transition relation η is defined as follows: $\xi \rightarrow \xi' \in \eta$ iff for some $i \in [0..n-1]$, for all $l \neq i$, $\xi(l) = \xi'(l)$, and for all $k \neq l$, $k \neq i$, $\xi(l, k) = \xi'(l, k)$, and

- $\xi'(i) \neq \xi(i)$, and a single incoming channel is updated by removal of a single letter and for a single outgoing channel such as $c_{i,k}$, e'_{ik} is added to the tail of $\xi(i,k)$. The following shows how the contents of an incoming channel, such as $c_{j,i}$, are updated:
  - if $a \in \beta'_{j,k}$ then $a\ \xi'(j,i) = \xi(j,i)$, or
  - if $a \in \beta'_{j,k}$, $r \in tpr(\Sigma_{j,i})$, and $\xi(j,i) = (a + a_1 + \ldots + a_m)^+ r$, then $\xi'(j,i) = \xi(j,i)$, or
  - if $a \in \beta'_{j,k}$, $r \in tpr(\Sigma_{j,i})$, and $\xi(j,i) = (a + a_1 + \ldots + a_m)^+ r$, then $\xi'(j,i) = r$, or
  - if $a \in \beta'_{j,k}$, $r \in tpr(\Sigma_{j,i})$, and $\xi(j,i) = (a + a_1 + \ldots + a_m)^+ r$, then $\xi'(j,i) = (a_1 + \ldots + a_m)^+ r$, or
  - if $a \in \beta'_{j,k}$, $r \in tpr(\Sigma_{j,i})$, and $\xi(j,i) = (a + a_1 + \ldots + a_m)^+ r$, then $\xi'(j,i) = (a_1 + \ldots + a_m)^+ (a + a_1 + \ldots + a_m)^+ r$.

or

- $\xi'(i) = \xi(i)$, and a set of incoming channels is updated by removal of a set of letters, and a set of outgoing channels, such as $c_{i,k}$, is updated by writing the corresponding letters; for $t \in tpr(\Sigma_{i,k})$, if $\xi(i,k) \neq t(\epsilon_k)^+$, then $\xi'(i,k) = \xi(i,k)\ (\epsilon_k)^+$, otherwise $\xi'(i,k) = \xi(i,k)$. The following shows how the contents of incoming channels, such as $c_{j,i}$, are updated:
  - if $b \in \beta_{j,i}$, $r \in tpr(\Sigma_{j,i})$, and $\xi(j,i) = b\ r$, then $\xi'(j,i) = r$, or
  - if $r \in tpr(\Sigma_{j,i})$, $\xi(j,i) = (b_1 + \ldots + b_u)^+ r$, and $\beta_{j,i} \cap \{b_1, \ldots, b_u\} \neq \emptyset$, then $\xi'(j,i) = \xi(j,i)$, or
  - if $r \in tpr(\Sigma_{j,i})$, $\xi(j,i) = (b_1 + \ldots + b_u)^+ r$, and $\beta_{j,i} \cap \{b_1, \ldots, b_u\} \neq \emptyset$, then $\xi'(j,i) = r$, or
  - if $r \in tpr(\Sigma_{j,i})$, $\xi(j,i) = (b_1 + \ldots + b_u)^+ r$, and $\{b_1, \ldots, b_u\} \setminus \beta_{j,i} = \{d_1, \ldots, d_v\}$, then $\xi(j,i) = (d_1 + \ldots + d_v)^+ r$, or
  - if $r \in tpr(\Sigma_{j,i})$, $\xi(j,i) = (b_1 + \ldots + b_u)^+ r$, and $\{b_1, \ldots, b_u\} \setminus \beta_{j,i} = \{d_1, \ldots, d_v\}$, then $\xi(j,i) = (d_1 + \ldots + d_v)^+ (b_1 + \ldots + b_u)^+ r$.

Definition 9. *A computation path in $ADSM(n)$ is a finite or infinite sequence denoted $\xi = \xi_0 \to \xi_1 \to \ldots$ where $\xi_0 = q^0$ and for all i, $\xi_i \to \xi_{i+1} \in \eta$.*

The $\mathcal{L}^*(ADSM(n))$ is a subset of $(Q \times T)^*$ and consists of all the finite computations in $ADSM(n)$ and $\mathcal{L}^\omega(ADSM(n))$ is a subset of $(Q \times T)^\omega$ and consists of all the infinite and fair computations in $ADSM(n)$. The language of $ADSM(n)$, denoted by $\mathcal{L}(ADSM(n))$, is equal to $\mathcal{L}^*(ADSM(n)) \cup \mathcal{L}^\omega(ADSM(n))$.

Example: Two Automata with Two Channels. This example illustrates how the *tpr*'s are updated in the calculation of the abridged model of a $DSM(2)$ when there are no changes in the component states.

Let $A_1 = ((\Sigma, P, \delta_1, p^0), \leq_1)$ and $A_2 = ((\Sigma, Q, \delta_2, q^0), \leq_2)$ be two piecewise automata. Let $\Sigma = \Sigma_{1,2} \cup \Sigma_{2,1}$. The composite system of the two automata A_1 and A_2 with two channels is defined as $DSM(2) = (P \times Q, \{c_{1,2}, c_{2,1}\}, \Sigma, \Sigma_{1,2}^* \times \Sigma_{2,1}^*, (p^0, q^0, \lambda, \lambda), \delta)$. For $i \in [1..u]$, assume on the states from which process A_1 and A_2 do not have any transitions to other states there are a set of self loop conditional transitions in A_1: $c_{2,1}?d_i \to c_{1,2}!a_i$ and a set of self loop read and write transitions in A_2: $c_{1,2}?b_i$ and $c_{2,1}!e_i$, respectively.

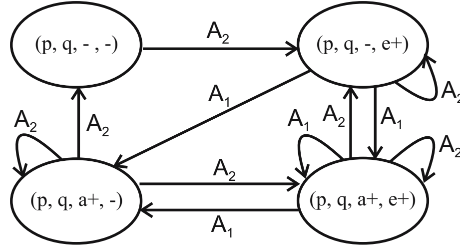


Fig. 1. Partial representation of global states in $ADSM(2)$ if $\alpha = \beta$ and $\gamma = \zeta$

Consider the computation path $(p^0, q^0, r^0, s^0) \rightarrow (p^1, q^1, r^1, s^1) \rightarrow \ldots$ in $ADSM(2)$ where $r^i \in tpr(\Sigma_{1,2})$ and $s^i \in tpr(\Sigma_{2,1})$. Because of the ordering relations on P and Q, for some i and for all j, $i < j$, $p^j = p^i$ and $q^j = q^i$.

Let $\alpha = \{a \in \Sigma_{1,2} \mid \text{for some } d \in \Sigma_{2,1},\ \delta_1(p^i, (d, a)) = p^i\} = \{a_1, \ldots, a_k\}$, and $\beta = \{b \in \Sigma_{1,2} \mid \delta_2(q^i, b) = q^i\} = \{b_1, \ldots, b_l\}$, $\gamma = \{d \in \Sigma_{2,1} \mid \text{for some } a \in \Sigma_{1,2},\ \delta_1(p^i, (d, a)) = p^i\} = \{d_1, \ldots, d_m\}$, and $\zeta = \{e \in \Sigma_{2,1} \mid \delta_2(q^i, e) = q^i\} = \{e_1, \ldots, e_n\}$. First assume $r^i = \lambda$ and $s^i = \lambda$. Let $a = \langle +\alpha \rangle$, $b = \langle +\beta \rangle$, $d = \langle +\gamma \rangle$, and $e = \langle +\zeta \rangle$. Figure 1 shows the possible transitions in the mentioned path starting from $(p^i, q^i, \lambda, \lambda)$ supposing $\alpha = \beta$ and $\gamma = \zeta$. Since the component state (p^i, q^i) stays the same, the superscript i is not shown in the figure. Symbol λ is also shown by '$-$' symbol.

As the figure shows, according to the transition relation of $ADSM(n)$, a write operation performed by process A_2 causes a transition from global state (p, q, λ, λ) to (p, q, λ, e^+). Process A_2 can continue writing on channel $c_{2,1}$ or process A_1 can read from $c_{2,1}$ and write on $c_{1,2}$. In the latter case, channel $c_{2,1}$ may become empty after some reads, depicted by a transition to state (p, q, a^+, λ), or it may still contain some letters, depicted by a transition to state (p, q, a^+, e^+). Figure 2 illustrates the possible transitions supposing $\beta \subset \alpha$, $\alpha - \beta = \epsilon$, and $\gamma = \zeta$. Let $f = \langle +\epsilon \rangle$. The state machine corresponding to the case where $\gamma \subset \zeta$ can be constructed similarly. If $s^i \neq \lambda$ then there are only two cases to consider. A_2 can only add at most one atomic expression to s^i, namely $(e_1 + \ldots + e_n)^+$. Furthermore, A_1 can only decrease the length of s^i, whether to λ or not. In both cases there are only a finite number of ancestors.
End of Example.

For the global state $(q_0, \ldots, q_{n-1}, t_{0,1}, \ldots, t_{n-1,n-2})$ we use notation (q, t) in order to represent its component state and channel contents. The following lemma shows that the next state relation of $ADSM(n)$ is finite.

Lemma 1. *If* $DSM(n) = (Q, C, \Sigma, R, q^0, \delta)$ *and* $ADSM(n) = (Q, T, q^0, \eta, \Phi)$ *is the abridged model of* $DSM(n)$*, given* $(q, t) \in Q \times T$*, the set of* $(q', t') \in Q \times T$ *such that* $(q, t) \rightarrow (q', t') \in \eta$ *is finite.*

Proof: This follows from the finiteness of the A_i's, Σ, and the definition of the transition relation η. □

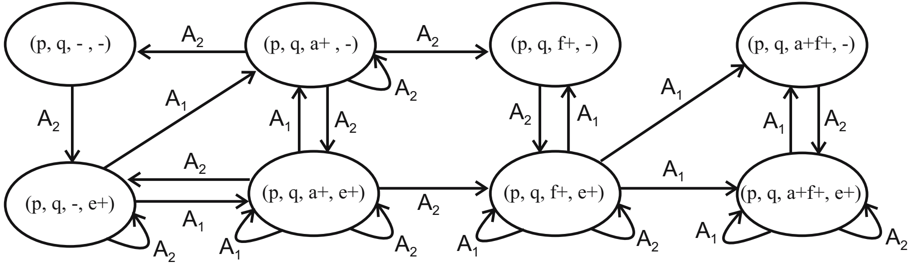


Fig. 2. Partial representation of global states in $ADSM(2)$ if $\beta \subset \alpha$ and $\gamma = \zeta$

Given $DSM(n)$, its abridged model, $ADSM(n)$, is constructed recursively: for each global state such as ξ we calculate the set $\{\xi' \mid \xi \rightarrow \xi' \in \eta\}$. Let G be the current set of global states $\{\xi\}$ of $ADSM(n)$ plus the transition connections between them. G represents that portion of $ADSM(n)$ that has been calculated so far.

Create Z which will consist of the component states of G plus sets of letters replacing the *tpr*'s of G. Set Z is used as a test of termination for the procedure of calculating G, i.e. it helps to determine whether in the further calculation of G a global state with a new component state is going to appear or not. Create the initial set of letters from a *tpr* as follows: start at the head of the *tpr* representing the content of channel $c_{i,j}$. Include in this set, all those letters in the head of the *tpr* for which A_j has a transition. If the *tpr* has no such letters then stop — A_j has no more inputs from channel $c_{i,j}$. As long as A_j has a transition for some letters in the head of the *tpr*, remove the head of the *tpr* and continue calculating the set of letters from the now shorter *tpr*. This process clearly terminates, either because the *tpr* is now empty or there is a limit on the different possible reads that can be performed from that channel by A_j.

Apply the above process for all channels and component states. Then update Z according to the transition relation of $DSM(n)$. If a conditional transition is triggered, e.g. A_i reads an a and then writes b, record this by adding b to the appropriate outgoing channel. If any of these transitions result in a new component state, then stop. The calculation of $ADSM(n)$ is not finished yet — destroy Z and continue calculating G, i.e. continue calculating $ADSM(n)$. Otherwise, continue to calculate Z. This process terminates since no letters are ever removed from the set associated to the *tpr* representing the contents of the channels for any given component state of Z. Therefore, these sets will stop growing after a finite number of updates.

If in the process of updating Z none of the transitions result in a new component state and the sets of letters stop growing, it is implicit that no more global states with new component states will appear in the further calculation of G.

Theorem 1. *If* $DSM(n) = (Q, C, \Sigma, R, q^0, \delta)$, *the procedure for generating the abridged model* $ADSM(n) = (Q, T, q^0, \eta, \Phi)$ *terminates.*

Proof: In order to prove the termination of the procedure of generating $ADSM(n)$, we have to prove that the process of calculating G terminates.

A new Z is only created if any of the transitions result in a new component state. This only happens a finite number of times since there are only so many component state configurations, given that the states of any single piecewise automaton come with a partial order that is respected by the transition relation of $DSM(n)$ and therefore of $ADSM(n)$. Thus, termination of the calculation of Z implies that the calculation of G has reached a point from which there are no more global states with new component states left to be explored. □

In the procedure of calculating $ADSM(n)$, after calculating all the global states with distinct component states, any new global state is only created through updates to *tpr*'s that represent the contents of the channels. Consider channel $c_{k,i}$. Assume state s is a state from which process A_k does not have any transitions to other states. Let $\epsilon_i = \langle +e_{ki} \mid$ for some b at the head of A_k's incoming channels, $s \in \delta_k(s, (b, e_{ki}))\rangle$. Process A_k can only update the contents of $c_{k,i}$ by appending ϵ_i to the tail of its associated *tpr*. Since there is a limit on the different possible (conditional) writes on $c_{k,i}$ that can be performed by A_k, there will be a finite number of ϵ_i's. Process A_i can only decrease the length of the *tpr* that represents the contents of the channel $c_{k,i}$. This only happens a finite number of times. Thus, there will be a finite number of updates to the *tpr*'s in the further calculation of the global states of $ADSM(n)$. This was also illustrated in the previous example.

Let $\psi = \psi_0 \rightarrow \psi_1 \rightarrow \ldots$ be a computation in $DSM(n)$ and $\xi = \xi_0 \rightarrow \xi_1 \rightarrow \ldots$ be a computation in $ADSM(n)$. ξ and ψ are two *corresponding* computations if for all i, $\psi_0(i) = \xi_0(i)$ and for every global state with distinct component state in ψ, such as ψ_k, there exists a corresponding global state in ξ, ξ_g, such that for all i, $\psi_k(i) = \xi_g(i)$. In other words, the component states of the corresponding global states should be identical. In addition, the order of the appearance of the corresponding global states in two corresponding computations should be the same.

Lemma 2. *For every computation in $DSM(n)$, there exists a corresponding computation in its abridged model, $ADSM(n)$, and for every computation in $ADSM(n)$, there exists a set of corresponding computations in $DSM(n)$.*

4 Automated Analysis

The main reason for abridging $DSM(n)$ is to be able to reason about its infinite computations by analyzing the computations of the finite $ADSM(n)$. According to the construction procedure of $ADSM(n)$, all the appropriate channel contents in $DSM(n)$ are represented by a set of *tpr*'s in $ADSM(n)$. If ψ_k and ξ_g are two corresponding global states of $DSM(n)$ and $ADSM(n)$ respectively, and the channel contents are given by $r = (\psi_k(0,1), \psi_k(0,2), \ldots, \psi_k(n-1, n-2))$ and $t = (\xi_g(0,1), \xi_g(0,2), \ldots, \xi_g(n-1, n-2))$, then $r \in t$ denotes that each $\psi_k(i,j)$ is contained in $\xi_g(i,j)$.

Lemma 3. *For every reachable state in $DSM(n)$, (q, r), there exists a reachable state in $ADSM(n)$, (q, t), where $r \in t$ and for every reachable state in $ADSM(n)$, (q, t), there exists a reachable state in $DSM(n)$, (q, r), where $r \in t$.*

As explained, the component states in the global states of $ADSM(n)$ are composed of the reachable component states in $DSM(n)$. Since $ADSM(n)$ is a finite state system, the reachability analysis can be performed by an exhaustive search of its state space.

The computations of $ADSM(n)$ satisfy property S if and only if there is no computation x of $ADSM(n)$ such that $ADSM(n), x \models \neg S$.

We use a standard automata theoretic technique to decide this problem [24]. The technique consists of creating an automaton, $\mathcal{B}_{\neg S}$, on infinite strings, *cf.* [11] and [18], which accepts only those strings that satisfy the property $\neg S$. We combine the structure $ADSM(n)$ with $\mathcal{B}_{\neg S}$ to form the product automaton $ADSM(n) \times \mathcal{B}_{\neg S}$. This is an automaton on infinite strings whose language is empty if and only if the computations of $ADSM(n)$ satisfy the property S.

A Büchi automaton is an automaton that recognizes infinite strings. A Büchi automaton over the alphabet Σ is of the form $\mathcal{B} = (Q, q^0, \Delta, F)$ with finite state set Q, initial state $q^0 \in Q$, transition relation $\Delta \subseteq Q \times \Sigma \times Q$, and a set $F \subseteq Q$ of accepting states. A run of $\mathcal{B}$ on a ω-word $\alpha = \alpha(0)\alpha(1)\ldots$ from Σ^ω is a sequence $\sigma(0)\sigma(1)\ldots$ such that $\sigma(0) = q^0$ and $(\sigma(i), \alpha(i), \sigma(i+1)) \in \Delta$ for $i \geq 0$. The run is called accepting if it satisfies the fairness constraint F, i.e. some state of F occurs infinitely often on the run. $\mathcal{B}$ accepts α if there is an accepting run of $\mathcal{B}$ on α. The ω-language recognized by $\mathcal{B}$ is denoted as $L(\mathcal{B}) = \{\alpha \in \Sigma^\omega \mid \mathcal{B}$ accepts $\alpha\}$.

We consider system properties expressed by a restricted class of Büchi automata. Here, a Büchi automaton for $DSM(n)$ has a fixed set of possible states, at most one for each component state in $DSM(n)$. Since the set of computations in $ADSM(n)$ is a superset of the set of computations in $DSM(n)$, we require that the language of each Büchi property automaton be stuttering closed.

Lemma 4. *If $\mathcal{B}$ is a stuttering closed Büchi property automaton for $DSM(n)$, for every computation in $\mathcal{L}^\omega(ADSM(n))$ and the language of the property automaton, $\mathcal{L}(\mathcal{B})$, there exists a corresponding computation in $\mathcal{L}^\omega(DSM(n))$ and $\mathcal{L}(\mathcal{B})$ and for every computation in $\mathcal{L}^\omega(DSM(n))$ and $\mathcal{L}(\mathcal{B})$ there exists a corresponding computation in $\mathcal{L}^\omega(ADSM(n))$ and $\mathcal{L}(\mathcal{B})$.*

5 Analysis of an IP-Telecommunication Architecture

BoxOS is AT&T's virtual telecommunication network based on IP [7, 15, 25]. In this architecture, a telephone call is presented by a set of boxes representing telephones and call features that communicate over possibly unbounded, perfect communication channels. At a sufficient level of abstraction, each box represents a finite state transducer.

Figure 3 describes part of a transparent box that represents a communication template that all telephony features should implement. The transparent box communicates with two neighbors across four separate channels. Messages to/from the upstream (initiating), *caller*, are sent/received via i channels. Messages to/from the downstream (receiving), *callee*, are sent/received via o channels. Importantly, the language of the automaton that models the behavior of

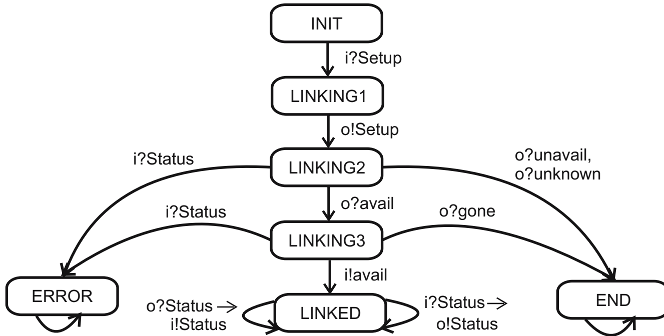


Fig. 3. Template feature box

the transparent box (as shown in Figure 3) is a piecewise regular language. It should be noted that a telephone call may be represented by a $DSM(n)$ and Figure 3 depicts one of the piecewise automata in the telephone call.

In our framework, different sets of properties can be specified that establish the correct behavior of the transparent box. A safety property can be verified by solving a reachability problem based on the negation of the safety property, for example, checking the reachability of a dedicated error state. A class of end-to-end temporal system properties specify that, for instance, if a message is sent from one end, it will eventually be received at the other end; *Always*(Send ⇒ *Eventually* Receive) whose negation is: *Eventually*(Send and *Always* not Received)[22]. For example, if a setup message is sent from the initiating *caller*, it will eventually be received by the *callee*. Another class of round-trip properties ensure that for every request there will eventually be a reply. For example, if a *caller* places a call (sends a setup message) and does not tear it down, eventually it receives one of the outcome signals 'unknown', 'unavail', or 'avail' from downstream.

It is worth noting that representing the contents of the channels by *tpr*'s allows specification of a wider range of channel properties, such as the existence of a specific message in a channel. A more thorough analysis of these properties is left for future work.

6 Summary and Future Work

We have presented an automated procedure for analyzing properties of FIFO systems of piecewise processes that occur naturally in the description of IP-telecommunication architectures. Further, it is evident that communication protocols must be 'well-designed' or satisfy some similar notion. Such distributed systems are prone to errors and our analysis techniques can be used to check for the presence of errors. For the future, we are interested in incorporating our analysis technique into standardized analysis tools and developing extensions of our techniques that are applicable to non-piecewise models.

References

1. P. A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *Proc. of TACAS'99*, LNCS 1579, pages 208–222, 1999.
2. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *Proc. of LICS'93*, pages 160–170, 1993.
3. P. Argon, G. Delzanno, S. Mukhopadhyay, and A. Podelski. Model checking for communication protocols. In *Proc. of SOFSEM'01*, LNCS 2234, 2001.
4. B. Boigelot. *Symbolic method for exploring infinite states spaces.* PhD thesis, Université de Liège, 1998.
5. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. *FMSD*, 14(3):237–255, 1999.
6. B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. of the 4th Int. Symp. on Static Analysis*, LNCS 1302, pages 172 – 186, 1997.
7. G. W. Bond, F. Ivančić, N. Klarlund, and R. Trefler. Eclipse feature logic analysis. *Second IP Telephony Workshop*, 2001.
8. A. Bouajjani, B. Jonsson, M. Nillson, and T. Touili. Regular model checking. In *Proc. of CAV'00*, LNCS, 2000.
9. A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *Proc. of LICS'01*, 2001.
10. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
11. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. of Int. Cong. on Logic, Methodology, and Philosophy of Science*, 1960.
12. G. Cece, A. Finkel, and S. P. Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1):20–31, 1996.
13. A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.
14. A. Finkel and L. Rosier. A survey on the decidability questions for classes of FIFO nets. In *Advances in Petri Nets*, LNCS 340, pages 106–132. Springer, 1988.
15. M. Jackson and P. Zave. Distributed Feature Composition: A virtual architecture for telecommunications services. *IEEE Trans. on Soft. Eng.*, 24(10):831–847, 1998.
16. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theo. Comp. Sci.*, 256(1–2):93–112, 2001.
17. N. Klarlund and R. Trefler. Regularity results for fifo channels. In *Proc. of AVoCS'04*, 2004. To appear in Electronic Notes in Theoretical Computer Science.
18. M. Nivat and D. Perrin. *Automata on Infinite Words.* Springer Verlag, 1985.
19. J. K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. *Proc. of PSTV*, pages 207–219, 1987.
20. J.-E. Pin. Syntactic semigroups. In Rozenberg and Salomaa, editors, *Handbook of language theory, Vol. I.* Springer Verlag, 1997.
21. J.-E. Pin and P. Weil. Polynomial closure and unambiguous product. *Theory Comput. Systems*, 30:1–39, 1997.
22. A. Pnueli. The temporal logic of programs. *Proc. 18th FOCS*, pages 46–57, 1977.
23. I. Simon. Piecewise testable events. *Proc. of 2nd GI Conf.*, 33:214–222, 1975.
24. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS'86*, pages 332–344, 1986.
25. P. Zave and M. Jackson. *The DFC Manual.* AT&T, 2001. Updated as needed. Available from `http:// www.research.att.com/ projects/ dfc`.

Ranking Abstraction of Recursive Programs*

Ittai Balaban[1], Ariel Cohen[1], and Amir Pnueli[1,2]

[1] Dept. of Computer Science, New York University
[2] Dept. of Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel

Abstract. We present a method for model-checking of safety and liveness properties over procedural programs, by combining state and ranking abstractions with procedure summarization. Our abstraction is an augmented finitary abstraction [KP00, BPZ05], meaning that a concrete procedural program is first augmented with a well founded ranking function, and then abstracted by a finitary state abstraction. This results in a procedural abstract program with strong fairness requirements which is then reduced to a finite-state *fair discrete system* (FDS) using procedure summarization. This FDS is then model checked for the property.

1 Introduction

Procedural programs with unbounded recursion present a challenge to symbolic model-checkers since they ostensibly require the checker to model an unbounded call stack. In this paper we propose the integration of ranking abstraction [KP00, BPZ05], finitary state abstraction, procedure summarization [SP81], and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. The inputs to this method are a sequential procedural program together with state and ranking abstractions. The output is either "success", or a counterexample in the form of an abstract error trace. The method is sound, as well as complete, in the sense that for any valid property, a sufficiently accurate *joint* (state and ranking) abstraction exists that establishes its validity.

The method centers around a fixpoint computation of procedure summaries of a finite-state program, followed by a subsequent construction of a behaviorally equivalent nondeterministic procedure-free program. Since we begin with an infinite-state program that cannot be summarized automatically, a number of steps involved in abstraction and LTL model-checking need to be performed over the procedural (unsummarized) program. These include augmentation with non-constraining observers and fairness constraints required for LTL verification and ranking abstraction, as well as computation of state abstraction. Augmentation with global observers and fairness is modeled in such a way as to make the associated properties observable once procedures are summarized. In computing the abstraction, the abstraction of a procedure call is handled by abstracting "everything but" the call itself, i.e., local assignments and binding of actual parameters to formals and of return values to variables.

* This research was supported in part by NSF grant CCR-0205571, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 267–281, 2006.
© Springer-Verlag Berlin Heidelberg 2006

The method relies on machinery for computing abstraction of first order formulas, but is orthogonal as to how abstraction is actually computed. We have implemented a prototype based on the TLV symbolic model-checker [Sha00] by extending it with a model of procedural programs. Specifically, given a symbolic finite-state model of a program, summaries are computed using BDD techniques in order to derive a *fair discrete system* (FDS) free of procedures to which model-checking is applied. The tool is provided, as input, with a concrete program and predicates and ranking components. It computes *predicate abstraction* [GS97] automatically using the method proposed in [BPZ05]. We have used this implementation to verify a number of canonical examples, such as Ackerman's function, the Factorial function and a procedural formulation of the 91 function.

While most components of the proposed method have been studied before, our approach is novel in that it reduces the verification problem to that of symbolic model-checking. Furthermore, it allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is supported directly by the model and related algorithms, rather than it being specified in a property.

1.1 Related Work

Recent work by Podelski et al. [PSW05] generalizes the concept of summaries to capture effects of computations between arbitrary program points. This is used to formulate a proof rule for total correctness of recursive programs with nested loops, in which a program summary is the auxiliary proof construct (analogous to an inductive invariant in an invariance proof rule). The rule and accompanying formulation of summaries represent a framework in which abstract interpretation techniques and methods for ranking function synthesis can be applied. In this manner both [PSW05] and our work aim at similar objectives. The main difference from our work is that, while we strive to work with abstraction of predicates, and use relations (and their abstraction) only for the treatment of procedures, the general approach of [PSW05] is based on the abstraction of relations even for the procedure-less case. A further difference is that, unlike our work, [PSW05] does not provide an explicit algorithm for the verification of aribtrary LTL properties. Instead it relies on a general reduction from proofs of termination to LTL verification.

Recursive State Machines (RSMs) [AEY01, ABE+05] and Extended RSMs [ACEM05] enhance the power of finite state machines by allowing for the recursive invocation of state machines. They are used to model the control flow of programs containing recursive procedure calls, and to analyze reachability and cycle detection. They are, however, limited to programs with finite data. On the other hand, the method that we present in this paper can be used to verify recursive programs with infinite data domains by making use of ranking and finitary state abstractions.

In [BR00], an approach similar to ours for computing summary relations for procedures is implemented in the symbolic model checker Bebop. However, while Bebop is able to determine whether a specific program statement is reachable, it cannot prove termination of a recursive boolean program or of any other liveness property.

The paper is organized as follows: In Section 2 we present the formal model of (procedure-free) fair discrete systems, and model-checking of LTL properties over them.

Section 3 formalizes recursive procedural programs presented as flow-graphs. In Section 4 we present a method for verifying the termination of procedural programs using ranking abstraction, state abstraction, summarization, construction of a procedure-free FDS, and finally, model-checking. In Section 5 we present a method for LTL model-checking of recursive procedural programs. Finally, Section 6 concludes and discusses future work.

2 Background

2.1 Fair Discrete Systems

The computation model, *fair discrete systems* (FDS) $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, consists of the following components:

- V : A finite set of *variables*. We define a *state* s to be an interpretation of the variables in V. Denoted by Σ is the set of all states of V.
- Θ : The *initial condition*. It is an assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$.
- $\mathcal{J}$: A set of *justice (weak fairness)* requirements (assertions).
- $\mathcal{C}$: A set of *compassion (strong fairness)* requirements (assertions). Each compassion requirement is a pair $\langle p, q \rangle$ of state assertions.

A *run* of an FDS is a sequence of states $\sigma : s_0, s_1, ...$, satisfying the following:

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For every $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

A *computation* of an FDS is an infinite run which also satisfies:

- *Justice:* For every $J \in \mathcal{J}$, σ contains infinitely many states satisfying J.
- *Compassion:* For every $\langle p, q \rangle \in \mathcal{C}$, σ should include either only finitely many p-states, or infinitely many q-states.

An FDS $\mathcal{D}$ is said to be *feasible* if it has at least one computation.

A *synchronous parallel composition* of systems $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathcal{D}_1 \,|||\, \mathcal{D}_2$, is specified by the FDS $\mathcal{D} : \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, where

$$\begin{array}{lll} V = V_1 \cup V_2, & \rho = \rho_1 \wedge \rho_2, & \Theta = \Theta_1 \wedge \Theta_2, \\ \mathcal{J} = \mathcal{J}_1 \cup \mathcal{J}_2, & \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 & \end{array}$$

Synchronous parallel composition is used for the construction of an observer system $\mathcal{O}$, which evaluates the behavior of another system $\mathcal{D}$. That is, running $\mathcal{D} \,|||\, \mathcal{O}$ will allow $\mathcal{D}$ to behave as usual while $\mathcal{O}$ evaluates it.

2.2 Linear Temporal Logic – LTL

LTL is an extension of propositional logic with two additional basic temporal operators, $\bigcirc$ (Next) and $\mathcal{U}$ (Until), from which $\diamondsuit$ (Eventually), $\square$ (Always), and $\mathcal{W}$ (Waiting-

for) can be derived. An LTL formula is a combination of assertions using the boolean operators $\neg$ and $\wedge$ and the temporal operators:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi$$

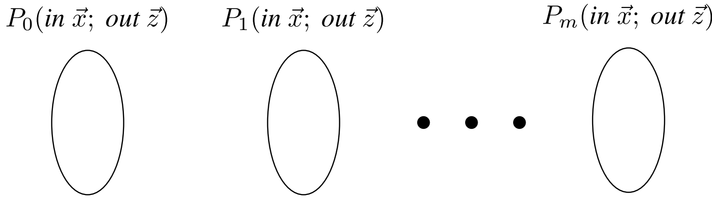
An LTL formula φ is *satisfied* by computation σ, denoted $\sigma \models \varphi$, if φ holds at the initial state of σ. An LTL formula φ is $\mathcal{D}$*-valid*, denoted $\mathcal{D} \models \varphi$, if all the computations of an FDS $\mathcal{D}$ satisfy φ.

Every LTL formula φ is associated with a *temporal tester*, an FDS denoted by $T[\varphi]$. A tester contains a distinguished boolean variable x such that for every computation σ of $T[\varphi]$, for every position $j \geq 0$, $x[s_j] = 1 \iff (\sigma, j) \models \varphi$. This construction is used for model-checking an FDS $\mathcal{D}$ in the following manner:

- Construct a temporal tester $T[\neg\varphi]$ which is initialized with $x = 1$, i.e. an FDS that comprises just those computations that falsify φ.
- Form the synchronous parallel composition $\mathcal{D} \parallel\mid T[\neg\varphi]$, i.e. an FDS for which all of its computations are of $\mathcal{D}$ and which violate φ.
- Check feasibility of $\mathcal{D} \parallel\mid T[\neg\varphi]$. $\mathcal{D} \models \varphi$ if and only if $\mathcal{D} \parallel\mid T[\neg\varphi]$ is infeasible.

3 Recursive Programs

A program P consists of $m+1$ modules: $P_0, P_1, \ldots, P_m$, where P_0 is the main module, and $P_1, \ldots, P_m$ are procedures that may be called from P_0 or from other procedures.



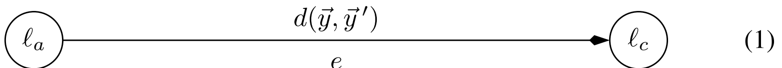
Each module P_i is presented as a flow-graph with its own set of locations $\mathcal{L}_i = \{\ell_0^i, \ell_1^i, \ldots, \ell_t^i\}$. It must have ℓ_0^i as its only entry point, ℓ_t^i as its only exit, and every other location must be on a path from ℓ_0^i to ℓ_t^i. It is required that the entry node has no incoming edges and that the terminal node has no outgoing edges.

The variables of each module P_i are partitioned into $\vec{y} = (\vec{x}; \vec{u}; \vec{z})$. We refer to $\vec{x}, \vec{u}$, and $\vec{z}$ as the input, working (local), and output variables, respectively. A module cannot modify its own input variables.

3.1 Edge Labels

Each edge in the graph is labeled by an instruction that has one of the following forms:

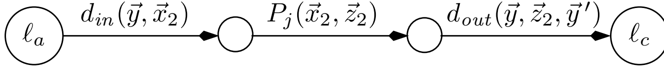
- A *local change* $d(\vec{y}, \vec{y}')$, where d is an assertion over two copies of the module variables.



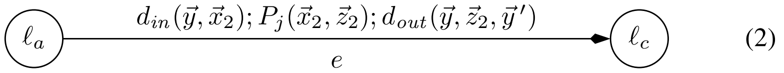
(1)

It is required that $d(\vec{y}, \vec{y}')$ implies $\vec{x}' = \vec{x}$.

- A *procedure call* $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$, where $\vec{x}_2$ and $\vec{z}_2$ are fresh copies of the input and output parameters $\vec{x}$ and $\vec{z}$, respectively.



This instruction represents a procedure call to procedure P_j where several elements are non-deterministic. The assertion $d_{in}(\vec{y}, \vec{x}_2)$ determines the actual arguments that are fed in the variables of $\vec{x}_2$. It may also contain an enabling condition under which this transition is possible. The assertion $d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ updates the module variables $\vec{y}$ based on the values returned by the procedure P_j via the output parameters $\vec{z}_2$. It is required that $d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ implies $\vec{x}\,' = \vec{x}$. Unless otherwise stated, we shall use the following description as abbreviation for a procedure call.



(2)

Example 1 (The 91 Function). Consider the functional program specified by

$$F(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } F(F(x + 11)) \tag{3}$$

We refer to this function as F_{91}. Fig. 1 shows the procedural version of F_{91}. In the figure, as well as subsequent examples, the notation $\vec{v}_1 := f(\vec{v}_2)$ denotes $\vec{v}_1' = f(\vec{v}_2) \wedge$ *pres*$(\vec{y} - \vec{v}_2)$, with *pres*$(\vec{v})$ defined as $\vec{v}' = \vec{v}$, for some set of variables $\vec{v}$. ▟

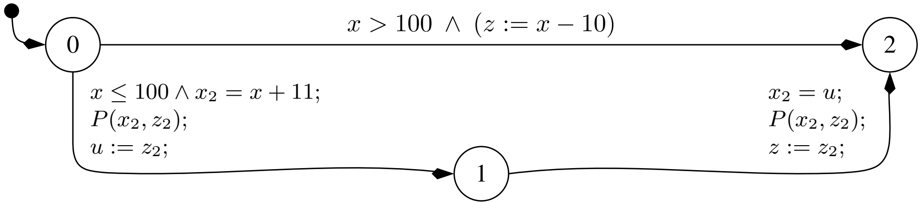


Fig. 1. Procedural program F_{91}

3.2 Computations

A *computation* of a program P is a maximal (possibly infinite) sequence of states and their labeled transitions:

$$\sigma : \langle \ell_0^0; (\xi, \vec{\bot}, \vec{\bot}) \rangle \xrightarrow{\lambda_1} \langle \ell^1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_2} \langle \ell^2; \vec{\nu}_2 \rangle \cdots$$

where each $\vec{\nu}_i = (\xi_i, \eta_i, \zeta_i)$ is an interpretation of the variables $(\vec{x}, \vec{u}, \vec{z})$. The values $\vec{\bot}$ denote uninitialized values. Labels in the transitions are either names of edges in the program or the special label *return*. Each transition $\langle \ell; \vec{\nu} \rangle \xrightarrow{\lambda} \langle \ell'; \vec{\nu}' \rangle$ in a computation must be justified by one of the following cases:

Assignment: There exists an assignment edge e of the form presented in Diagram (1), such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_c$ and $\langle \vec{\nu}, \vec{\nu}' \rangle \models d(\vec{y}, \vec{y}')$.

Procedure Call: There exists a call edge e of the form presented in Diagram (2), such that $\ell = \ell_a$, $\lambda = e$, $\ell' = \ell_0^j$, and $\vec{\nu}' = (\xi', \vec{\bot}, \vec{\bot})$, where $\langle \vec{\nu}, \xi' \rangle \models d_{in}(\vec{y}, \vec{x}_2)$.

Return: There exists a procedure P_j (the procedure from which we return), such that $\ell = \ell_t^j$ (the terminal location of P_j). The run leading up to $\langle \ell; \vec{\nu} \rangle$ has a suffix of the form

$$\langle \ell_1; \vec{\nu}_1 \rangle \xrightarrow{\lambda_1} \underbrace{\langle \ell_0^j; (\xi; \vec{\bot}; \vec{\bot}) \rangle \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_k} \langle \ell; (\xi; \eta; \zeta) \rangle}_{\sigma_1}$$

such that the segment σ_1 is *balanced* (has an equal number of *call* and *return* labels), $\lambda_1 = e$ is a call edge of the form presented in Diagram (2), where $\ell' = \ell_c$, $\lambda =$ *return*, and $\langle \vec{\nu}_1, \zeta, \vec{\nu}' \rangle \models d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$.
This definition uses the computation itself in order to retrieve the context as it were before the corresponding call to procedure P_j.

For a run $\sigma_1 : \langle \ell_0^0; (\xi, \vec{\bot}, \vec{\bot}) \rangle \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_k} \langle \ell; \vec{\nu} \rangle$, we define the *level* of state $\langle \ell; \vec{\nu} \rangle$, denoted *Lev*$(\langle \ell; \vec{\nu} \rangle)$, to be the number of "call" edges in σ_1 minus the number of "return" edges.

4 Verifying Termination

This section presents a method for verifying termination of procedural programs. Initially, the system is augmented with well-founded ranking components. Then a finitary state abstraction is applied, resulting in a finite-state procedural program. Procedure summaries are computed over the abstract, finite-state program, and a procedure-free FDS is constructed. Finally, infeasibility of the derived FDS is checked, showing that it does not possess a fair divergent computation. This establishes the termination of the original program.

4.1 A Proof Rule for Termination

The application of a ranking abstraction to procedures is based on a rule for proving termination of loop-free procedural programs. We choose a well founded domain $(\mathcal{D}, \succ)$, such that for each procedure P_i with input parameters $\vec{x}$, we associate a *ranking function* δ_i that maps $\vec{x}$ to $\mathcal{D}$. For each edge e in P_i, labeled by a procedure call as shown in Diagram (2), we generate the descent condition $D_e(\vec{y}) : d_{in}(\vec{y}, \vec{x}_2) \rightarrow \delta_i(\vec{x}) \succ \delta_j(\vec{x}_2)$. The soundness of this proof rule is stated by the following claim:

Claim 1 (Termination). If the descent condition $D_e(\vec{y})$ is valid for every procedure call edge e in a loop-free procedural program P, then P terminates.

Proof: (Sketch) A non-terminating computation of a loop-free program must contain a subsequence of the form

$$\langle \ell_0^0; (\xi_0, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_0}^0; (\xi_0, \eta_0, \zeta_0) \rangle, \langle \ell_0^{j_1}; (\xi_1, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_1}^{j_1}; (\xi_1, \eta_1, \zeta_1) \rangle,$$
$$\langle \ell_0^{j_2}; (\xi_2, \vec{\bot}, \vec{\bot}) \rangle, \ldots, \langle \ell_{i_2}^{j_2}; (\xi_2, \eta_2, \zeta_2) \rangle, \langle \ell_0^{j_3}; (\xi_3, \vec{\bot}, \vec{\bot}) \rangle, \ldots$$

where, for each $k \geq 0$, *Lev*$(\langle \ell_0^{j_k}; (\xi_k, \vec{\bot}, \vec{\bot}) \rangle) =$ *Lev*$(\langle \ell_{i_k}^{j_k}; (\xi_k, \eta_k, \zeta_k) \rangle) = k$. If the descent condition is valid for all call edges, this leads to the existence of the infinitely descending sequence

$$\delta_0(\xi_0) \succ \delta_{j_1}(\xi_1) \succ \delta_{j_2}(\xi_2) \succ \delta_{j_3}(\xi_3) \succ \cdots$$

which contradicts the well-foundedness of the δ_i's. ∎

Space limitations disallow a proof of the following completeness result:

Claim 2 (Completeness). The method of proving termination is complete for loop-free programs.

Validity of the condition D_e is to be interpreted semantically. Namely, $D_e(\vec{y})$ should hold for every $\vec{\nu}$, such that there exists a computation reaching location ℓ_a with $\vec{y} = \vec{\nu}$.

4.2 Ranking Augmentation of Procedural Programs

Ranking augmentation was suggested in [KP00] and used in [BPZ05] in conjunction with predicate abstraction to verify liveness properties of non-procedural programs. In its application here we require that a ranking function be applied only over the input parameters. Each procedure is augmented with a *ranking observer* variable that is updated at every procedure call edge e, in a manner corresponding to the descent condition D_e. For example, if the observer variable is *inc* then a call edge

$$d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2; \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}')$$

is augmented to be

$$d_{in}(\vec{y}, \vec{x}_2) \wedge \mathit{inc}' = \mathit{sign}(\delta(\vec{x}_2) - \delta(\vec{x}));\ P_j(\vec{x}_2; \vec{z}_2);\ d_{out}(\vec{y}, \vec{z}_2, \vec{y}') \wedge \mathit{inc}' = 0$$

All local assignments are augmented with the assignment $\mathit{inc} := 0$, as the ranking does not change locally in a procedure. Well foundedness of the ranking function is captured by the compassion requirement $(\mathit{inc} < 0, \mathit{inc} > 0)$ which is being imposed only at a later stage.

Unlike the termination proof rule, the ranking function need not decrease on every call edge. Instead, a program can be augmented with multiple similar components, and it is up to the feasibility analysis to sort out their interaction and relevance automatically.

Example 2 (Ranking Augmentation of Program F_{91}). We now present an example of ranking abstraction applied to program F_{91} of Fig. 1. As a ranking component, we take

$$\delta(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } 0 \textbf{ else } 101 - x$$

Fig. 2 presents the program augmented by the variable *inc*. ∎

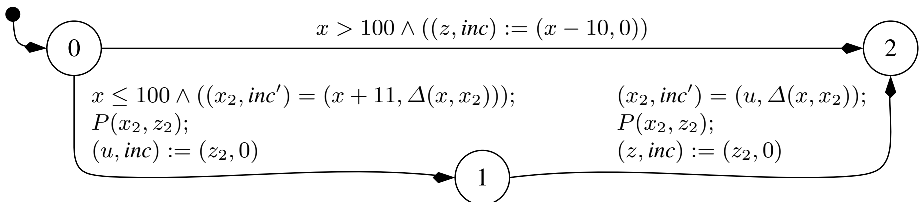


Fig. 2. Program F_{91} augmented by a Ranking Observer. The notation $\Delta(x_1, x_2)$ denotes the expression $\mathit{sign}(\delta(x_2) - \delta(x_1))$.

4.3 Predicate Abstraction of Augmented Procedural Programs

We consider the application of finitary abstraction to procedural programs, focusing on predicate abstraction for clarity. We assume a predicate base that is partitioned into $\vec{T} = \{\vec{I}(\vec{x}), \vec{W}(\vec{y}), \vec{R}(\vec{x},\vec{z})\}$, with corresponding abstract (boolean) variables $\vec{b}_T = \{\vec{b}_I, \vec{b}_W, \vec{b}_R\}$. For each procedure the input parameters, working variables, and output parameters are $\vec{b}_I$,$\vec{b}_W$, and $\vec{b}_R$, respectively.

An abstract procedure will have the same control-flow graph as its concrete counterpart, where only labels along the edges are abstracted as follows:

- A local change relation $d(\vec{y}, \vec{y}\,')$ is abstracted into the relation

$$D(\vec{b}_T, \vec{b}'_T): \quad \exists \vec{y}, \vec{y}\,'.\, \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}'_T = \vec{T}(\vec{y}\,') \wedge d(\vec{y}, \vec{y}\,')$$

- A procedure call $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,')$ is abstracted into the abstract procedure call $D_{in}(\vec{b}_T, \vec{b}^2_I); P_j(\vec{b}^2_I, \vec{b}^2_R); D_{out}(\vec{b}_T, \vec{b}^2_R, \vec{b}'_T)$, where

$$\begin{array}{ll} D_{in}(\vec{b}_T, \vec{b}^2_I) & : \exists \vec{y}, \vec{x}_2.\, \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}^2_I = \vec{I}(\vec{x}_2) \wedge d_{in}(\vec{y}, \vec{x}_2) \\ D_{out}(\vec{b}_T, \vec{b}^2_R, \vec{b}'_T) & : \exists \vec{y}, \vec{x}_2, \vec{z}_2, \vec{y}\,' \left(\begin{array}{l} \vec{b}_T = \vec{T}(\vec{y}) \wedge \vec{b}^2_R = \vec{R}(\vec{x}_2, \vec{z}_2) \wedge \vec{b}'_T = \vec{T}(\vec{y}\,') \wedge \\ d_{in}(\vec{y}, \vec{x}_2) \wedge d_{out}(\vec{y}, \vec{z}_2, \vec{y}\,') \end{array} \right) \end{array}$$

Example 3 (Abstraction of Program F_{91}).
We apply predicate abstraction to program F_{91} of Fig. 1. As a predicate base, we take

$$\vec{I}: \{x > 100\}, \qquad \vec{W}: \{u = g(x + 11)\}, \qquad \vec{R}: \{z = g(x)\}$$

where

$$g(x) \quad = \quad \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } 91$$

The abstract domain consists of the corresponding boolean variables $\{B_I, B_W, B_R\}$. The abstraction yields the abstract procedural program $P(B_I, B_R)$ which is presented in Fig. 3. ▟

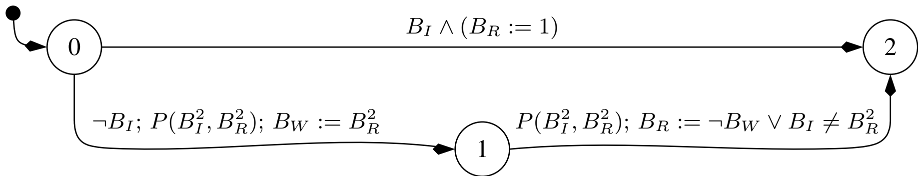


Fig. 3. An abstract version of Program F_{91}

Finally we demonstrate the joint (predicate and ranking) abstraction of program F_{91}.

Example 4 (Abstraction of Ranking-Augmented Program F_{91}).
We wish to abstract the augmented program from Example 2. When applying the abstraction based on the predicate set

$$\vec{I}: \{x > 100\}, \qquad \vec{W}: \{u = g(x + 11)\}, \qquad \vec{R}: \{z = g(x)\}$$

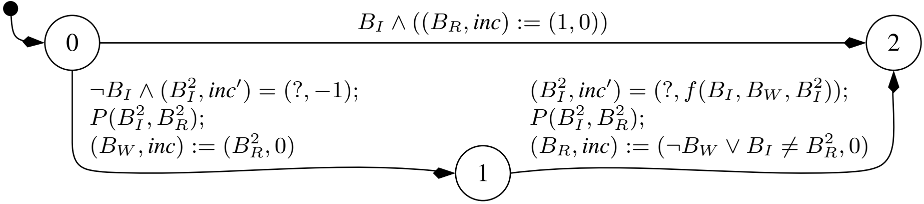


Fig. 4. An abstract version of Program F_{91} augmented by a Ranking Observer

we obtain the abstract program presented in Fig. 4, where

$$f(B_I, B_W, B_I^2) \quad = \quad \textbf{if} \quad \neg B_I \wedge (B_I^2 \vee \neg B_I^2 \wedge B_W) \ \textbf{then} \ -1 \quad \textbf{else if} \ B_I \wedge B_I^2 \ \textbf{then} \ 0 \quad \textbf{else} \ 1$$

Note that some (in fact, all) of the input arguments in the recursive calls are left non-deterministically 0 or 1. In addition, on return from the second recursive call, it is necessary to augment the transition with an adjusting assignment that correctly updates the local abstract variables based on the returned result.

It is interesting to observe that all terminating calls to this abstract procedure return $B_R = 1$, thus providing an independent proof that program F_{91} is partially correct with respect to the specification $z = g(x)$.

The analysis of this abstract program yields that $\neg B_I \wedge B_W$ is an invariant at location 1. Therefore, the value of $f(B_I, B_W, B_I^2)$ on the transition departing from location 1 will always be -1. Thus, it so happens that even without feasibility analysis, from Claim 1 we can conclude that the program terminates. ■

4.4 Summaries

A procedure *summary* is a relation between input and output parameters. A relation $q(\vec{x}, \vec{z})$ is a summary if it holds for any $\vec{x}$ and $\vec{z}$ iff there exists a run in which the procedure is called and returns, such that the input parameters are assigned $\vec{x}$ and on return the output parameters are assigned $\vec{z}$.

Since procedures may contain calls (recursive or not) to other procedures, deriving summaries involves a fixpoint computation. An *inductive assertion network* is generated that defines, for each procedure P_j, a summary q^j and an assertion φ_a^j associated with each location ℓ_a. For each procedure we construct a set of constraints according to the rules of Table 1. The constraint $\varphi_t^j(\vec{x}, \vec{u}, \vec{z}) \rightarrow q^j(\vec{x}, \vec{z})$ derives the summary from the assertion associated with the terminating location of P_j. All assertions, beside φ_0^j, are initialized false. φ_0^j, which refers to the entry location of P_j, is initialized true, i.e. it allows the input variables to have any possible value at the entry location of procedure P_j. Note that the matching constraint for an edge labeled with a call to procedure $P_i(\vec{x}_2; \vec{z}_2)$ encloses the summary of that procedure, i.e. the summary computation of one procedure comprises summaries of procedures being called from it.

An iterative process is performed over the constraints contributed by all procedures in the program, until a fixpoint is reached. Reaching a fixpoint is guaranteed since all variables are of finite type.

Table 1. Rules for Constraints contributed by Procedure P_j to the Inductive Assertion Network

Fact	**Constraint(s)**
	$\varphi_0^j = \mathsf{true}$ $\varphi_t^j(\vec{x},\vec{u},\vec{z}) \rightarrow q^j(\vec{x},\vec{z})$
$\ell_a \xrightarrow{d(\vec{y},\vec{y}\,')} \ell_c$	$\varphi_a^j(\vec{y}) \wedge d(\vec{y},\vec{y}\,') \rightarrow \varphi_c^j(\vec{y}\,')$
$\ell_a \xrightarrow{d_{in}(\vec{y},\vec{x}_2)} \ell_c$	$\varphi_a^j(\vec{y}) \wedge d_{in}(\vec{y},\vec{x}_2) \rightarrow \varphi_c^j(\vec{y},\vec{x}_2)$
$\ell_a \xrightarrow{P_i(\vec{x}_2;\vec{z}_2)} \ell_c$	$\varphi_a^j(\vec{y},\vec{x}_2) \wedge q^i(\vec{x}_2,\vec{z}_2) \rightarrow \varphi_c^j(\vec{y},\vec{z}_2)$
$\ell_a \xrightarrow{d_{out}(\vec{y},\vec{z}_2,\vec{y}\,')} \ell_c$	$\varphi_a^j(\vec{y},\vec{z}_2) \wedge d_{out}(\vec{y},\vec{z}_2,\vec{y}\,') \rightarrow \varphi_c^j(\vec{y}\,')$

Claim 3 (Soundness). Given a minimal solution to the constraints of Table 1, q_j is a summary of P_j, for each procedure P_j.

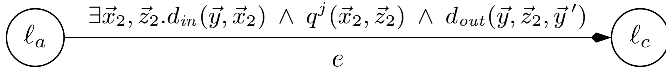
Proof. In one direction, let $\sigma : s_0, \ldots, s_t$ be a computation segment starting at location ℓ_0^j and ending at ℓ_t^j, such that $\vec{x}[s_0] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$. It is easy to show by induction on the length of σ that $s_t \models \varphi_t^j(\vec{x},\vec{u},\vec{z})$. From Table 1 we obtain $\varphi_t^j(\vec{x},\vec{u},\vec{z}) \rightarrow q^j(\vec{x},\vec{z})$. Therefore $s_t \models q^j(\vec{x},\vec{z})$. Since all edges satisfy $\vec{x} = \vec{x}\,'$, we obtain $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x},\vec{y})$.

In the other direction, assume $[\vec{x} \mapsto \vec{v}_1, \vec{y} \mapsto \vec{v}_2] \models q^j(\vec{x},\vec{y})$. From the constraints in Table 1 and the minimality of their solution, there exists a state s_t with $\vec{x}[s_t] = \vec{v}_1$ and $\vec{z}[s_t] = \vec{v}_2$ such that $s_t \models \varphi_t^j$. Repeating this reasoning we can, by propagating backward, construct a computation segment starting at ℓ_0 that initially assigns $\vec{v}_1$ to $\vec{x}$.

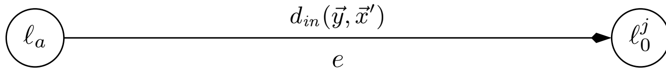
4.5 Deriving a Procedure-Free FDS

Using summaries of an abstract procedural program P_A, one can construct the *derived* FDS of P_A, labeled $\mathsf{derive}(P_A)$. This is an FDS denoting the set of *reduced computations* of P_A, a notion formalized in this section. The variables of $\mathsf{derive}(P_A)$ are partitioned into $\vec{x}, \vec{y}$, and $\vec{z}$, each of which consists of the input, working, and output variables of all procedures, respectively. The FDS is constructed as follows:

- Edges labeled by local changes in P_A are preserved in $\mathsf{derive}(P_A)$
- A procedure call in P_A, denoted by a sequence of edges of the form $d_{in}(\vec{y},\vec{x}_2); P_j(\vec{x}_2,\vec{z}_2); d_{out}(\vec{y},\vec{z}_2,\vec{y}\,')$ from a location ℓ_a to a location ℓ_c, is transformed into the following edges:
  - A *summary* edge, specified by



  - A *call* edge, specified by



- All compassion requirements, which are contributed by the ranking augmentation and described in Subsection 4.2, are imposed on $\mathsf{derive}(P_A)$.

The reasoning leading to this construction is that summary edges represent procedure calls that return, while call edges model non-returning procedure calls. Therefore, a summary edge leads to the next location in the calling procedure while modifying its variables according to the summary. On the other hand, a call edge connects a calling location to the entry location of the procedure that is being called. Thus, a nonterminating computation consists of infinitely many call edges, and a call stack is not necessary.

We now prove soundness of the construction. Recall the definition of a computation of a procedural program given in Subsection 3.2. A computation can be terminating or non-terminating. A terminating computation is finite, and has the property that every computation segment can be extended to a *balanced* segment, which starts with a calling step and ends with a matching return step. A computation segment is *maximally balanced* if it is balanced and is not properly contained in any other balanced segment.

Definition 1. *Let σ be a computation of P_A. Then the* reduction *of σ, labeled* $\mathsf{reduce}(\sigma)$, *is a sequence of states obtained from σ by replacing each maximal balanced segment by a summary-edge traversal step.*

Claim 4. For any sequence of states σ, σ is a computation of $\mathsf{derive}(P_A)$ iff there exists σ', a computation of P_A, such that $\mathsf{reduce}(\sigma') = \sigma$.

Proof of the claim follows from construction of $\mathsf{derive}(P_A)$ in a straightforward manner. It follows that if σ is a terminating computation of P_A, then $\mathsf{reduce}(P_A)$ consists of a single summary step in the part of $\mathsf{derive}(P_A)$ corresponding to P_0. If σ is an infinite computation of P_A, then $\mathsf{reduce}(\sigma)$ (which must also be infinite) consists of all assignment steps and calls into procedures from which σ has not returned.

Claim 5 (Soundness – Termination). If $\mathsf{derive}(P_A)$ is infeasible then P_A is a terminating program.

Proof. Let us define the notion of abstraction of computations. Let $\sigma = s_0, s_1, \ldots$ be a computation of P, the original procedural program from which P_A was abstracted. The abstraction of σ is a computation $\alpha(s_0), \alpha(s_1), \ldots$ where for all $i \geq 0$, if s_i is a state in σ, then $\alpha(s_i) = [\vec{b}_I \mapsto \vec{I}(\vec{x}), \vec{b}_W \mapsto \vec{W}(\vec{y}), \vec{b}_R \mapsto \vec{R}(\vec{x}, \vec{z})]$.

Assume that $\mathsf{derive}(P_A)$ is infeasible. Namely, every infinite run of $\mathsf{derive}(P_A)$ violates a compassion requirement. Suppose that P has an infinite computation σ. Consider $\mathsf{reduce}(\sigma)$ which consists of all steps in non-terminating procedure invocations within σ. Since the abstraction of $\mathsf{reduce}(\sigma)$ is a computation of $\mathsf{derive}(P_A)$ it must be unfair with respect to some compassion requirement. It follows that a ranking function keeps decreasing over steps in $\mathsf{reduce}(\sigma)$ and never increases – a contradiction. ∎

4.6 Analysis

The feasibility of $\mathsf{derive}(P_A)$ can be checked by conventional symbolic model-checking techniques. If it is feasible then there are two possibilities: (1) The original system truly diverges, or (2) feasibility of the derived system is *spurious*, that is, state and ranking abstractions have admitted behaviors that were not originally present. In the latter case, the method presented here can be repeated with a refinement of either state or ranking abstractions. The precise nature of such refinement is outside the scope of this paper.

5 LTL Model Checking

In this section we generalize the method discussed so far to general LTL model-checking. To this end we adapt to procedural programs the method discussed in Subsection 2.2 for model-checking LTL by composition with temporal testers [KPR98]. We prepend the steps of the method in Section 4 with a *tester composition step* relative to an LTL property. Once ranking augmentation, abstraction, summarization, and construction of the derived FDS are computed, the resulting system is model-checked by conventional means as to feasibility of initial states that do not satisfy the property.

The main issue is that synchronous composition of a procedural program with a global tester, including justice requirements, needs to be expressed in terms of local changes to procedure variables. In addition, since LTL is modeled over infinite sequences, the derived FDS needs to be extended with idling transitions.

5.1 Composition with Temporal Testers

A temporal tester is defined by a unique global variable, here labeled t, a transition relation $\rho(\vec{z}, t, \vec{z}\,', t')$[1] over primed and unprimed copies of the tester and program variables, where t does not appear in $\vec{z}$, and a justice requirement. In order to simulate global composition with ρ, we augment every procedure with the input and output parameters t_i and t_o, respectively, as follows:

- An edge labeled by a local change is augmented with $\rho(\vec{z}, t_o, \vec{z}\,', t_o')$
- A procedure call of the form $d_{in}(\vec{y}, \vec{x}_2); P_j(\vec{x}_2, \vec{z}_2); d_{out}(\vec{y}, \vec{x}_2, \vec{y}\,')$ is augmented to be $d_{in}(\vec{y}, \vec{x}_2) \ \wedge \ \rho(\vec{z}, t_o, \vec{x}_2, t_i^2); P_j((\vec{x}_2, t_i^2), (\vec{z}_2, t_o^2)); d_{out} \ \wedge \ \rho(\vec{z}_2, t_o^2, \vec{z}\,', t_o')$
- Any edge leaving the initial location of a procedure is augmented with $t_o = t_i$

Example 5. Consider the program in Fig. 5. Since this program does not terminate, we are interested in verifying the property $\varphi : (\Diamond\, z) \ \vee \ \Box \Diamond\, at_\ell_2$, specifying that either eventually a state with $z = 1$ is reached, or infinitely often location 2 of P_1 is visited. To verify φ we decompose its negation into its principally temporal subformulas, $\Box\, \neg z$ and $\Diamond \Box\, \neg at_\ell_2$, and compose the system with their temporal testers. Here we demonstrate the composition with $T[\Box\, \neg z]$, given by the transition relation $t = \neg z \wedge t'$ and the trivial justice requirement $\mathtt{true}$. The composition is shown in Fig. 6. ▪

As a side remark, we note that our method can handle global variables in the same way as applied for global variables of testers, i.e., represent every global variable by a set of input and output parameters and augment every procedure with these parameters and with the corresponding transition relations.

5.2 Observing Justice

In order to observe justice imposed by a temporal tester, each procedure is augmented by a pair of *observer* variables that consists of a working and an output variables. Let J be a justice requirement, P_i be a procedure, and the associated observer variables be J_u and J_o. P_i is augmented as follows: On initialization, both J_u and J_o are assigned

[1] We assume here that the property to be verified is defined over the output variable only.

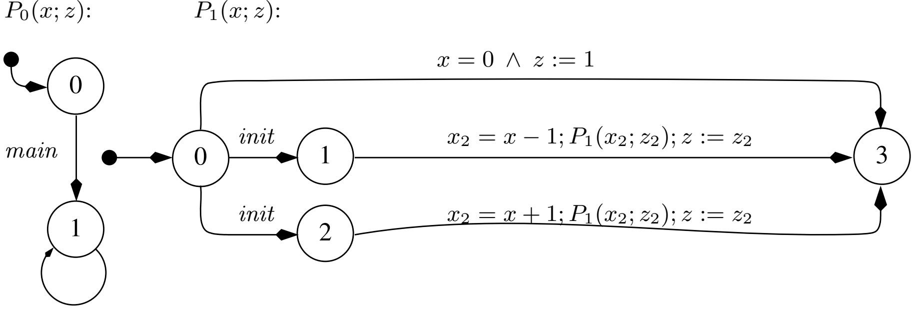


Fig. 5. A Divergent Program. *init* represents $x > 0 \wedge z := 0$, and *main* represents $x \geq 0 \wedge x_2 := x; P_1(x_2; z_2); z := z_2$.

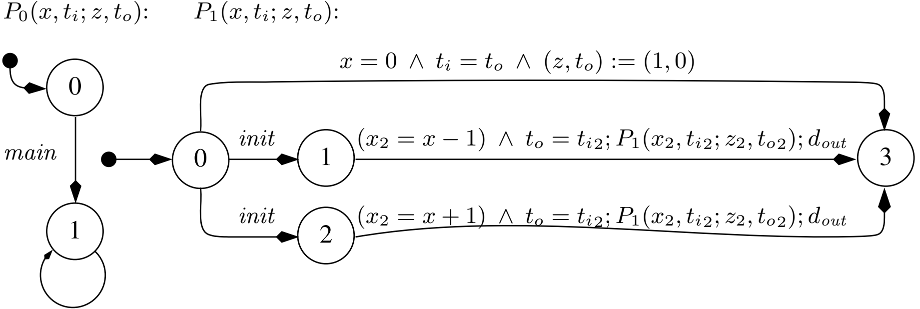


Fig. 6. The Program of Fig. 5, Composed with $T[\Box \neg z]$. The assertion d_{out} represents $t_{o2} = (\neg z_2 \wedge t'_o) \wedge z := z_2$, *init* represents $x > 0 \wedge t_i = t_o \wedge z := 0$, and *main* represents $x \geq 0 \wedge (x_2 = x) \wedge (t_o = \neg z \wedge t_{i2}); P_1(x_2, t_{i2}; z_2, t_{o2}); d_{out}$.

true if the property J holds at that state. Local changes are conjoined with $J_u := J'$ and $J_o := (J_o \vee J')$. Procedure calls are conjoined with $J_u := (J' \vee J_o^2)$ and $J_o := (J_o \vee J' \vee J_o^2)$, where J_o^2 is the relevant output observer variable of the procedure being called.

While J_u observes J at every location, once J_o becomes true it remains so up to the terminal location. Since J_o participates in the procedure summary, it is used to denote whether justice has been satisfied within the called procedure.

5.3 The Derived FDS

We use the basic construction here in deriving the FDS as in Section 4.5. In addition, for every non-output observer variable J_u we impose the justice requirement that in any fair computation, J_u must be true infinitely often. Since LTL is modeled over infinite sequences, we must also ensure that terminating computations of the procedural program are represented by infinite sequences. To this end we simply extend the terminal location of procedure P_0 with a self-looping edge. Thus, a terminating computation is one that eventually reaches the terminal location of P_0 and stays idle henceforth.

In this section we use the notation $\mathsf{derive}(P_A)$ to denote the FDS that is derived from P_A and thus extended. The following claim of soundness is presented without proof due to space limitations.

Claim 6 (Soundness – LTL). Let P be a procedural program, φ be a formula whose principal operator is temporal, and P_A be the abstract program resulting from the composition of P with the temporal tester $T[\neg\varphi]$ and its abstraction relative to a state and ranking abstraction. Let t_o be the tester variable of $T[\neg\varphi]$. If $t_o = \mathtt{true}$ is an infeasible initial state of $\mathsf{derive}(P_A)$ then φ is valid over P.

6 Conclusion

We have described the integration of ranking abstraction, finitary state abstraction, procedure summarization, and model-checking into a combined method for the automatic verification of LTL properties of infinite-state recursive procedural programs. Our approach is novel in that it reduces the verification problem of procedural programs with unbounded recursion to that of symbolic model-checking. Furthermore, it allows for application of ranking and state abstractions while still relegating all summarization computation to the model-checker. Another advantage is that fairness is being supported directly by the model, rather than being specified in a property.

We have implemented a prototype based on the TLV symbolic model-checker and tested several examples such as Ackerman's function, the Factorial function and a recursive formulation of the 91 function. We verified that they all terminate and model checked satisfiability of several LTL properties.

As further work it would be interesting to investigate concurrency with bounded context switching as suggested in [RQ05]. Another direction is the exploration of different versions of LTL that can relate to nesting levels of procedure calls, similar to the manner in which the CARET logic [AEM04] expresses properties of recursive state machines concerning the call stack.

References

[ABE$^+$05] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T.W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[AEM04] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *TACAS'04*, pages 467–481.

[ACEM05] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In *TACAS'05*, pages 61–765.

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV'01*, pages 207–220.

[BPZ05] I. Balaban, A. Pnueli, and L.D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, pages 164–180.

[BR00] T. Ball and S.K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'00*, pages 113–130.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97*, pages 72–83.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *CAV'98*, pages 1–16.

[PSW05] A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. In *ESOP'05*, pages 94–107.

[RQ05] J. Rehof and S. Qadeer. Context-bounded model checking of concurrent software. In *TACAS'05*, pages 93–107.

[Sha00] E. Shahar. *The TLV Manual*, 2000. http://www.cs.nyu.edu/acsys/tlv.

[SP81] M. Sharir and A. Pnueli. Two approaches to inter-procedural data-flow analysis. In Jones and Muchnik, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

Relative Safety

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing, National University of Singapore,
S16, 3 Science Drive 2, Singapore 117543, Republic of Singapore
{joxan, andrews, razvan}@comp.nus.edu.sg

Abstract. A safety property restricts the set of reachable states. In this paper, we introduce a notion of *relative safety* which states that certain program states are reachable provided certain other states are. A key, but not exclusive, application of this method is in representing *symmetry* in a program. Here, we show that relative safety generalizes the programs that are presently accommodated by existing methods for symmetry. Finally, we provide a practical algorithm for proving relative safety.

1 Introduction

A safety property restricts the set of reachable states. Let $[\![P]\!]$ denote the collecting semantics of a program P with variables $\tilde{X}$. Thus each sequence $\tilde{x}$ of variable values in $[\![P]\!]$ represents a reachable state. A safety property may be simply written as a constraint Ψ over the variables $\tilde{X}$. For example, the safety property $X+Y<9$ states that in all reachable states, the values of the program variables X and Y sum to less than 9. If we let the predicate $p(\tilde{x})$ be *true* just in case the sequence of values of program variables $\tilde{x}$ is in $[\![P]\!]$, then a safety property may be written in the form $p(\tilde{X}) \models \Psi$, for example, $p(X,Y) \models X+Y<9$.

In this paper, we introduce the notion of *relative safety*. Briefly and informally, this asserts that a certain state is reachable provided a certain other state is reachable. Note that this does not mean that these two states share a computation path. Specifically, consider the specification of states in the form $p(\tilde{X}) \wedge \Psi$. That is, we use the constraint Ψ to identify the set of solutions of Ψ which correspond to reachable states. Then our notion of relative safety simply relates two of these specifications in the following way: $p(\tilde{X}) \wedge \Psi \models p(\tilde{Y}) \wedge \Psi'$ where Ψ and Ψ' are constraints over $\tilde{X}, \tilde{Y}$. For example, $p(X_1,X_2) \models p(Y_1,Y_2) \wedge X_1 = Y_2 \wedge X_2 = Y_1$ (or more succinctly, $p(X_1,X_2) \models p(X_2,X_1)$) asserts that if the state (α,β) is reachable, then so is (β,α), for all values α and β. In other words, the observable values of the two program variables commute.

Relative safety can specify powerful structural properties of programs. The driving application we consider in this paper is that of verification with symmetry reduction. Symmetry has been widely employed for minimizing the search space in program verification. It is a reduction technique employed in Murφ [13] and SMC [21] model checkers among many others. Symmetry is often defined using automorphisms π on the symmetric objects. These induce an equivalence relation between program states. Efficiency in state exploration is hence achieved by only checking the representatives of the equivalence classes.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 282–297, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Let us take as an example a concurrent program with two almost identical processes, where process 1 updates the variables PC_1 and X_1, and process 2, PC_2 and X_2. Here PC_1 and PC_2 are process 1 and 2's program counters, respectively. Let us consider $(\alpha,\beta,\gamma,\delta)$ to be values of (PC_1,PC_2,X_1,X_2). Classic symmetry "exchanges" process 1 and 2, that is, $\pi((\alpha,\beta,\gamma,\delta)) = (\beta,\alpha,\delta,\gamma)$. A necessary condition for π to be an automorphism is that whenever $\tilde{x}$ is a reachable state, so is $\pi(\tilde{x})$. Such a relation between $\tilde{x}$ and $\pi(\tilde{x})$ can be logically represented as the relative safety assertion $p(PC_1,PC_2,X_1,X_2) \models p(PC_2,PC_1,X_2,X_1)$ where the predicate p, once again, represents the reachable states of the program. Below we show many more examples of symmetry, including ones that are not covered by existing techniques.

The main technical part of this paper is a proof method. In its most basic form, the method to prove the assertion $G_1 \models G_2$ checks that the set of states represented by the symbolic formula G_2 is reachable, whenever the set G_1 is reachable. This is done by the basic operation of "backward unfolding" the program's transition relation. A key element in our algorithm is the use of the principle of coinduction which is critical for termination of the unfolding process.

The paper is organized as follows. We discuss some related work in Section 2. We then formalize the program semantics and the proof method in the framework of *Constraint Logic Programming* (*CLP*) [14], for two main reasons. First, the logical framework of CLP is eminently suitable for the represntation of our concept of relative safety, and second, the established implementation technology of CLP systems allow us to perform unfolding operations efficiently. We introduce some preliminary CLP concepts in Section 3. Relative safety is then formally defined in Section 4. Here, we show via several examples, novel ways to realize symmetry. In addition to these, we will also show a non-symmetry example. Section 5 formally presents our algorithm. Finally, in Section 6, we demonstrate the use of our prototype implementation on some classes of programs in order to show the practical potential of our algorithm.

2 Related Work

Existing approaches define symmetry on syntactic considerations. In contrast, our notion of relative safety is based on semantics. An advantage is more flexibility in specifying a wide range of symmetry-like properties, including many that would not be considered a symmetry property by the existing methods. One example, shown later, is a mutual exclusion algorithm with priority between processes. We can handle a wider range than [7, 20], for example. Importantly, relative safety goes far beyond symmetry (and below, we demonstrate the property of serializability).

In more detail, symmetry is often defined as a transition-preserving equivalence [8, 3, 13, 9, 20], where an automorphism π, other than being a bijection on the reachable states, also satisfies that $(\tilde{x},\tilde{x}')$ is a transition iff $(\pi(\tilde{x}),\pi(\tilde{x}'))$ is. Another notion of equivalence used is bisimilarity [7], replacing the second condition with bisimilarity on the state graph. These stronger equivalences allows for the handling of larger class of properties beyond safety such as CTL* properties. However, stronger equivalence also means less freedom in handling symmetries on the collecting semantics, which we exploit further in this paper.

In [20], while still defining symmetry as transition-preserving equivalence, they attempt to handle systems which state graphs are not fully symmetric. The approach transforms the state graph into a fully symmetric one, while keeping annotation for each transition that has no correspondence in the original state graph. The graph with full symmetry is then reduced by equating automorphic states. This work is the most general and can reduce the state graph of even totally asymmetric programs, however, its application is limited to programs with syntacticly specified static transition priority.

Similar to the work of [20], prior works infer symmetry based on syntactic conditions, such as concurrent program with identical processes or syntactic restrictions on program statements and variable usage. These also include the *scalarset* approach of Murϕ [13], and the limitation to permutation of process identifiers in SMC model checker [21]. In contrast, our approach to prove symmetry semanticly for each program enables us to treat more programs where the semantics is symmetric although the syntax is not.

An application of our symmetry proof method has been demonstrated in the context of timed automata verification [16]. This paper presents a generalization and automation of the method.

There have been many works in the area of verification using CLP (see [11] for a non-exhaustive survey), partly because it is natural to express transition relations as CLP rules. Due to its ability in handling constraints, CLP has been notably used in verification of infinite-state systems [5, 10, 12, 17], although results for finite-state systems are also available [18, 19]. None of these works, however, deal with relative safety.

3 CLP Representation of Programs

We start by stipulating that each process in a concurrent program has the usual syntax of a deterministic imperative language, and communication occurs via shared variables. We also have a blocking primitive **await** (**b**) **s** where **b** is a boolean expression and **s** a program statement, which can be executed only when **b** holds. A *program* is a collection of a fixed number of processes. We provide the 2-process bakery algorithm in Figure 1 as an example. We display program points in angle brackets.

We now introduce *CLP programs*. CLP programs have a *universe of discourse* $\mathcal{D}$ which is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. They are used in two ways: in the base programming language to describe expressions and conditionals, and in user assertions, defined below. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

```
    while (true) do
⟨0⟩     t1 := t2 + 1
⟨1⟩     await (t1<t2 ∨ t2=0) skip
⟨2⟩     t1 := 0
    end
```

```
    while (true) do
⟨0⟩     t2 := t1 + 1
⟨1⟩     await (t2<t1 ∨ t1=0) skip
⟨2⟩     t2 := 0
    end
```

Fig. 1. Bakery-2

$p([0,0],T_1,T_2) \leftarrow T_1 = 0, T_2 = 0.$ `% init`
$p([1,P_2],T_1',T_2) \leftarrow p([0,P_2],T_1,T_2), T_1' = T_2 + 1.$ `% r1`
$p([2,P_2],T_1,T_2) \leftarrow p([1,P_2],T_1,T_2), (T_1 < T_2 \vee T_2 = 0).$ `% e1`
$p([0,P_2],T_1',T_2) \leftarrow p([2,P_2],T_1,T_2), T_1' = 0.$ `% x1`
$p([P_1,1],T_1,T_2') \leftarrow p([P_1,0],T_1,T_2), T_2' = T_1 + 1.$ `% r2`
$p([P_1,2],T_1,T_2) \leftarrow p([P_1,1],T_1,T_2), (T_2 < T_1 \vee T_1 = 0).$ `% e2`
$p([P_1,0],T_1,T_2') \leftarrow p([P_1,2],T_1,T_2), T_2' = 0.$ `% x2`

Fig. 2. CLP Representation of Bakery-2

An *atom*, is as usual, of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the $\tilde{t}$ a tuple of terms. The set $\{p(\tilde{d})\}$ where p ranges over the predicates and $\tilde{d}$ ranges over the tuples in $\mathcal{D}$ is called the *domain base* $\mathcal{B}$ of our CLP programs.

Now, a CLP program is a set of *rules*. A rule is an implication of the form $A \leftarrow \tilde{B}, \phi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint ϕ constitute the *body* of the rule. We say that a rule is a (constrained) *fact* if $\tilde{B}$ is the empty sequence.

Translating a user program P_0 into an appropriate CLP program P is in fact intuitively straightforward; we thus provide only an informal outline here. Our CLP rules corresponding to a transition of the program will be of the form

$$p(PC', X_1', X_2', \ldots, X_n') \leftarrow p(PC, X_1, X_2, \ldots, X_n), \phi.$$

Here, PC is a list representing the program counters in the k processes of P_0 before the transition. Its primed counterpart PC' represents the list after the transition. $X_1, X_2, \ldots, X_n$ and their primed counterparts represent the variables in P_0 before and after the transition, while ϕ is a constraint on all the variables. Note that as in the above rule, throughout this paper we often use a comma in place of $\wedge$ to denote conjunction. The above rule depicts a transition from rhs to lhs .

Example 1 (Bakery-2). Consider our 2-process bakery algorithm in Figure 1. Note that the point $\langle 2 \rangle$ indicates the critical section, and initially, `t1` = `t2` = 0. The CLP program in Figure 2 (the parts preceded by `%` are comments) is in fact its CLP representation. x

The semantics of a CLP program is based on the concept of ground instances. A *ground instance* of a constraint ϕ is obtained by instantiating the variables therein from $\mathcal{D}$, and the result is true or false. We write this as $\phi\sigma$ [14] where $\sigma : var(\phi) \mapsto \mathcal{D}$ a *grounding*. Similarly, a *ground instance* of an atom or rule is obtained by instantiating variables therein with values in $\mathcal{B}$ using a grounding σ. Now consider the fixpoint operator $T_P : 2^{\mathcal{B}} \mapsto 2^{\mathcal{B}}$ for a CLP program P defined as follows: a ground atom $A\sigma$ is in $T_P(S)$ if $A\sigma \in S$ or there is a ground instance $(A \leftarrow \tilde{B}, \phi)\sigma$ of a rule $A \leftarrow \tilde{B}, \phi$ in P such that $\tilde{B}\sigma \subseteq S$ and $\phi\sigma$ is true. A basic theorem of CLP is that the least fixpoint of T_P is the least model of P, and this is also equal to the set of ground atoms. We denote this set by $[\![P]\!]$. A ground instance $A\sigma$ is true iff $A\sigma \in [\![P]\!]$. Similarly, a ground instance $(\tilde{B}, \phi)\sigma$ of a goal is true iff $\tilde{B}\sigma \subseteq [\![P]\!]$ and $\phi\sigma$ is true. We denote the set of true ground instances of a goal G by $[\![G]\!]$.

In general, where P is the CLP representation of P_0, we have that the collecting semantics of P_0 is characterized by $[\![P]\!]$.

4 Relative Safety

We now present an assertion language to express relative safety property, and demonstrate its expressive power for program reasoning. We start with a definition of a *constraint state*.

Definition 1 (Constraint State). *A* constraint state *is a goal in the form* $p(PC, X_1, \ldots, X_n), \phi$ *where* $PC, X_1, \ldots, X_n$ *represent the list of program counters and program variables, and* ϕ *is a constraint on the variables.*

Now let G^L be a constraint state and G^R either constraint or constraint state. Let $\tilde{X} = var(G^L) \cup var(G^R)$.

Definition 2 (Relative Safety). *A* relative safety assertion *is of the form* $G^L \models G^R$. *Its meaning is* $\forall \tilde{X} : G^L \rightarrow G^R$ *that is, for each grounding* σ *such that* $G^L\sigma \in [\![G^L]\!]$, $G^R\sigma \in [\![G^R]\!]$.

Intuitively, a relative safety assertion specifies that certain states are reachable only if certain other states are.

Here we start with a traditional safety property, generally of the form:

$$p(PC, \tilde{X}),\ \phi \ \models \ \phi'$$

where ϕ and ϕ' are constraints on the program counter array PC and program variables $\tilde{X}$. For example, in the Bakery-2 program, the following assertions specify mutual exclusion.

$$p([P_1, P_2], T_1, T_2) \models P_1 \neq 2 \wedge P_2 \neq 2, \text{ or, } p([2,2], T_1, T_2) \models \mathit{false}$$

Now consider a relative safety assertion, stating symmetry for Bakery-2:

$$p([P_1, P_2], T_1, T_2) \models p([P_2, P_1], T_2, T_1).$$

Note that an automorphism must be included in a group with the composition of automorphisms as its operator [23]. Such a group is known as an *automorphism group*. Our idea is to use a set of relative safety assertions to specify possible automorphisms on reachable states. Note that a single relative safety assertion in general only describes a partial mapping, while an automorphism is total. In general we need a set of assertions to describe a total mapping π. Moreover, equivalence between states is obtained by also proving a complete set of assertions which represent the mappings in an automorphism group. This would include inverses, which proof is often straightforward. Suppose that $map(G^L \models G^R)$ is the mapping represented by the assertion $G^L \models G^R$. Now, as an example, the above symmetry assertion for Bakery-2 characterizes an automorphism group *Aut* on the collecting semantics as follows:

- We include the obvious $map(p([P_1, P_2], T_1, T_2) \models p([P_1, P_2], T_1, T_2))$ in *Aut* satisfying the existence of identity.
- By simple renaming $\{P_1 \mapsto P_2, P_2 \mapsto P_1, T_1 \mapsto T_2, T_2 \mapsto T_1\}$ on the above assertion, the reverse $map(p([P_2, P_1], T_2, T_1) \models p([P_1, P_2], T_1, T_2))$ is in *Aut* satisfying the existence of inverse.

– It is straightforward to show that if $map(G_1 \models G_2) \in Aut$ and $map(G_2 \models G_3) \in Aut$ then $map(G_1 \models G_3) \in Aut$.

We will prove the assertion later in Section 5. We now proceed with several examples.

Example 2 (Rotational Symmetry). Next we demonstrate *rotational* symmetry in the solution of N dining philosophers' problem using $N-1$ tickets. For simplicity, we assume there are N=3 philosophers having ids 1, 2 and 3, and there are 3 forks represented as boolean array `f`, where `f[1]`, `f[2]`, `f[3]` are forks between philosopher 3 and 1, 1 and 2, and 2 and 3, respectively. Initially the ticket number `t`=2. To save space, we do not show the actual code. For our purpose it is suffice to demonstrate the rotational symmetry as the assertion:

$$p([P_1,P_2,P_3],F_1,F_2,F_3,T) \models p([P_3,P_1,P_2],F_3,F_1,F_2,T),$$

where P_i denotes the program point of philosopher i, F_1, F_2 and F_3 are the values of `f[`i`]`, $1 \leq i \leq 3$, respectively, and T is the number of tickets left. The above assertion specifies a cyclic shift. For this example, arbitrary transposition does not result in automorphism.

Example 3 (Permutation of Variable-Value Pair). In [16] we discussed a timed automata version of Fischer's algorithm, a timing-based mutual exclusion algorithm. The pseudocode can be found in [1] and is not presented here to save space. The algorithm uses a global variable `k` whose value is the process identifier of the process that is about to enter the critical section. This is translated into a variable K in our CLP representation (also not shown here). Since the example uses timing, our CLP representation for the 2-process version uses the variables T_1 and $T2$, denoting the running time of each process. Our symmetry assertion here is

$$p([P_1,P_2],T_1,T_2,K) \models p([P_2,P_1],T_2,T_1,K'),\phi,$$

where ϕ constrains (K,K') to $(0,0),(1,2)$ or $(2,1)$. This is called *permutation of variable-value pair* [20] since it maps the value of a variable onto a new one without exchanging it with another variable. This is not covered by some previous approaches such as [13, 21].

Example 4 (Priority Mutual Exclusion). We can also express the kind of "approximate" symmetry, as exemplified by the simple 2-process priority mutual exclusion in Figure 3. Each process has ⟨2⟩ as the critical section. Initially, the values of both `x1` and `x2` are 0. We show the CLP representation in Figure 4. This example is semantically similar to the

```
    while (true) do
⟨0⟩     await (x2 = 0) x1 := 1
⟨1⟩     skip
⟨2⟩     x1 := 0
    end
```

```
    while (true) do
⟨0⟩     x2 := 1
⟨1⟩     await (x1 = 0) skip
⟨2⟩     x2 := 0
    end
```

Fig. 3. Priority Mutual Exclusion

$p([0,0],0,0).$
$p([1,P_2],1,X_2) \leftarrow p([0,P_2],X_1,X_2), X_2=0.$
$p([2,P_2],X_1,X_2) \leftarrow p([1,P_2],X_1,X_2).$
$p([0,P_2],0,X_2) \leftarrow p([2,P_2],X_1,X_2).$
$p([P_1,1],X_1,1) \leftarrow p([P_1,0],X_1,X_2).$
$p([P_1,2],X_1,X_2) \leftarrow p([P_1,1],X_1,X_2), X_1=0.$
$p([P_1,0],X_1,0) \leftarrow p([P_1,2],X_1,X_2).$

Fig. 4. CLP Representation of Priority Mutual Exclusion

asymmetric readers-writers in [6] and the priority mutual exclusion in [20]. Although the state graph of the program is not symmetric, the state space, i.e. the set of nodes in the state graph, is, and knowing this is already useful to prove safety properties such as mutual exclusion. We can represent the symmetry on the state space simply as:

$$p([P_1,P_2],X_1,X_2) \models p([P_2,P_1],X_2,X_1).$$

It is not immediately obvious that the program is symmetric based on syntactic observation alone.

Example 5 (Szymanski's Algorithm). Szymanski's algorithm is a more complex priority-based mutual exclusion algorithm which is commonly encountered in the literature. We show the pseudocode in Figure 5. Its CLP representation is in Figure 6.

Roughly speaking, since the algorithm is based on prioritizing Process 1 to enter the critical section ⟨8⟩, it is not possible for Process 2 to be in the critical section while Process 1 is at its trying section. For example, the following does not hold:

$$p([8,7],X_1,X_2) \models p([7,8],X_2,X_1).$$

It is because the program points $[8,7]$ are reachable while $[7,8]$ are not. In other words, there is a grounding for the lhs goal, but no grounding for the rhs goal. Therefore, a simple symmetry assertion such the one given in the bakery algorithm does not hold. However, the following "not-quite" symmetry assertions still hold:

$$p([8,P_2],X_1,X_2), P_2<3 \models p([P_2,8],X_2,X_1).$$
$$p([8,P_2],X_1,X_2), P_2>7 \models p([P_2,8],X_2,X_1).$$
$$p([9,P_2],X_1,X_2), P_2 \neq 7 \models p([P_2,9],X_2,X_1).$$
$$p([P_1,P_2],X_1,X_2), P_1 \neq 8, P_1 \neq 9 \models p([P_2,P_1],X_2,X_1).$$

```
     while (true) do
⟨0⟩      x1:=1
⟨1⟩      await(x2<3) skip
⟨2⟩      x1:=3
⟨3⟩      if (x2=1) do
⟨4⟩          x1:=2
⟨5⟩          await(x2=4) skip
         end
⟨6⟩      x1:=4
⟨7⟩      skip
⟨8⟩      await(x2<2∨x2>3) skip
⟨9⟩      x1:=0
     end

     while (true) do
⟨0⟩      x2:=1
⟨1⟩      await(x1<3) skip
⟨2⟩      x2:=3
⟨3⟩      if (x1=1) do
⟨4⟩          x2:=2
⟨5⟩          await(x1=4) skip
         end
⟨6⟩      x2:=4
⟨7⟩      await(x1<2) skip
⟨8⟩      skip
⟨9⟩      x2:=0
     end
```

Fig. 5. 2-Process Szymanski's Algorithm

```
p([0,0],0,0). % Initial State
% Rules for Process 1
p([1,P2],1,X2)←p([0,P2],X1,X2).
p([2,P2],X1,X2)←p([1,P2],X1,X2),X2 < 3.
p([3,P2],3,X2)←p([2,P2],X1,X2).
p([4,P2],X1,X2)←p([3,P2],X1,X2),X2 = 1.
p([5,P2],2,X2)←p([4,P2],X1,X2).
p([6,P2],X1,X2)←p([3,P2],X1,X2),X2 ≠ 1.
p([6,P2],X1,X2)←p([5,P2],X1,X2).
p([7,P2],4,X2)←p([6,P2],X1,X2).
p([8,P2],X1,X2)←p([7,P2],X1,X2).
p([9,P2],X1,X2)←p([8,P2],X1,X2),
    (X2 < 2∨X2 > 3).
p([0,P2],0,X2)←p([9,P2],X1,X2).

% Rules for Process 2
p([P1,1],X1,1)←p([P1,0],X1,X2).
p([P1,2],X1,X2)←p([P1,1],X1,X2),X1 < 3.
p([P1,3],X1,3)←p([P1,2],X1,X2).
p([P1,4],X1,X2)←p([P1,3],X1,X2),X1 = 1.
p([P1,5],X1,2)←p([P1,4],X1,X2).
p([P1,6],X1,X2)←p([P1,3],X1,X2),X1 ≠ 1.
p([P1,6],X1,X2)←p([P1,5],X1,X2).
p([P1,7],X1,4)←p([P1,6],X1,X2).
p([P1,8],X1,X2)←p([P1,7],X1,X2),X1 < 2.
p([P1,9],X1,X2)←p([P1,8],X1,X2).
p([P1,0],X1,0)←p([P1,9],X1,X2).
```

Fig. 6. CLP Representation of Szymanski's Algorithm

At first it seems that the above assertions no longer defines an automorphism group since $p([P_1,8],X_1,X_2), 3 \leq P_1 \leq 7 \models p([8,P_1],X_2,X_1)$ can be derived from the last assertion, yet the inverse does not hold. However, by observation the assertion $p([P_1,8], X_1,X_2) \models P_1 < 3 \vee P_1 > 7$ holds since it is not possible for process 2 to be in the critical section while process 1 is waiting. Similarly, $p([P_1,9],X_1,X_2) \models P_1 \neq 7$ also holds. These impose restrictions on the last assertion above.

We are not aware of any verification technique that would allow us to express and use this kind of symmetry.

Example 6 (Serializability). We next discuss an application of relative safety assertion beyond symmetry. We show a producer/consumer program in Figure 7, which CLP representation is in Figure 8. The macros `con`$_k$`()` and `pro`$_l$`()`, abstract program fragments that serve to produce and consume respectively. We will imagine that apart from the variable `full` there are other variables `x` which may be used in `con`$_k$`()` and `pro`$_l$`()`.

Consider the assertions:

$$p([n+1,P_2],Full,f(X)),P_2 \leq n \models p([1,P_2],Full,X).$$
$$p([P_1,n],Full,g(X)),P_1 \geq 1 \models p([P_1,0],Full,X).$$

where the expression $f(X)$ and $g(X)$ are the results of performing `con`$_1$`()` ... `con`$_n$`()` and `pro`$_1$`()` ... `pro`$_n$`()` respectively on X. Then the assertions say that the *result* of performing the interleaving of `con`$_k$`()` and `pro`$_l$`()` macros, $1 \leq k \leq P_1 - 1$, $1 \leq l \leq P_2$ is as though the two sequences of transitions are serializable. Note that here we still have an automorphism group which contains the above assertions and their inverses.

Both symmetry and serializability are examples of *non-behavioral* properties, i.e., properties determined by the structure of the program. They are not necessarily related to the intended result of the computation. Relative safety is potentially useful to specify many other useful non-behavioral properties, possibly ad-hoc and application specific. The class of such properties is potentially large. It is intuitively clear that such information can help in speeding up the proof process of other properties, which we will demonstrate later.

```
Consumer:
     while (true) do
⟨0⟩       await (full=1) full:=0
⟨1⟩       con₁() ⟨2⟩ ... ⟨n⟩ conₙ() ⟨n+1⟩
     end
```

```
Producer:
     while (true) do
⟨0⟩       pro₁() ⟨1⟩ ... ⟨n−1⟩ proₙ()
⟨n⟩       await (full=0) full:=1 ⟨n+1⟩
     end
```

Fig. 7. Producer/Consumer

$p([0,0],0,X)$. % Initial State

% Consumer
$p([1,P_2],0,X) \leftarrow p([0,P_2],1,X)$.
$p([2,P_2],Full,X) \leftarrow p([1,P_2],Full,X)$.
$p([n,P_2],Full,X) \leftarrow p([n-1,P_2],Full,X)$.
$p([0,P_2],Full,X) \leftarrow p([n,P_2],Full,X)$.

% Producer
$p([P_1,1],Full,X) \leftarrow p([P_1,0],Full,X)$.
$p([P_1,n],Full,X) \leftarrow p([P_1,n-1],Full,X)$.
$p([P_1,n+1],Full,X) \leftarrow p([P_1,n],Full,X)$.
$p([P_1,0],1,X) \leftarrow p([P_1,n+1],0,X)$.

Fig. 8. Partial CLP Representation of Producer/Consumer

5 The Proof Method

Now let $G = (B_1,\dots,B_n,\phi)$ and P denote a goal and program respectively. Let $R = A \leftarrow C_1,\dots,C_m,\phi_1$ denote a rule in P, written so that none of its variables appear in G. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of G using R, denoted by $\text{reduct}(G,R)$, is of the form

$$(B_1,\dots,B_{i-1},C_1,\dots,C_m,B_{i+1},\dots,B_n,B_i = A \wedge \phi \wedge \phi_1)$$

provided the constraint $B_i = A \wedge \phi \wedge \phi_1$ has a true ground instance. Since the CLP rules are implications, it follows that $G \leftarrow \text{reduct}(G,R)$ holds.

Definition 3 (Unfold). *Given a program P and a goal G which contain one atom, a complete unfold of a goal G, denoted by* $\text{unfold}(G)$ *is the set* $\{G' | \exists R \in P : G' = \text{reduct}(G,R)\}$. *A (not necessarily complete) unfold of G is a set* $\text{unfold}'(G) \subseteq \text{unfold}(G)$.

Note that since $[\![G]\!] \neq \emptyset$ only if $G \cap T_P([\![\bigvee \text{unfold}(G)]\!]) \neq \emptyset$, and this holds only if $[\![\bigvee \text{unfold}(G)]\!] \neq \emptyset$, we have the *logical semantics of unfold*: $G \rightarrow \bigvee \text{unfold}(G)$.

Definition 4 (Unfold Tree Goals). *Given a program P and a set H of goals each contain one atom, we define the function $\delta(H) = H \cup \text{unfold}'(G_1)$, when $G_1 \in H$. We obtain a set of* unfold tree goals *of G by a finite successive applications of δ on $\{G\}$.*

Since for any goal G, $G \leftarrow \text{reduct}(G,R)$, for any goal G_1 in the unfold tree goals of G, $G_1 \rightarrow G$.

Definition 5 (Frontier). *Given a program P and a set H of goals which contains one atom, when there exists $G_1 \in H$, we define the nondeterministic function $\varepsilon(H) = (H - \{G_1\}) \cup \text{unfold}(G_1)$. $\varepsilon()$ can be successively applied to a singleton set containing an initial goal G obtaining a* frontier $F = \varepsilon(\dots(\varepsilon(\{G\}))\dots)$.

From the logical semantics of unfold, for any frontier F of G, $G \rightarrow \bigvee F$.

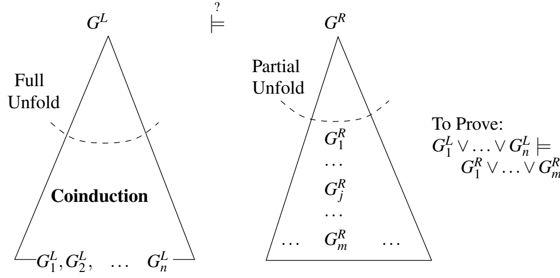


Fig. 9. Informal Structure of Proof Process

Intuitively, in order to prove $G^L \models G^R$, we proceed as follows: unfold G^L completely to obtain a frontier containing the goals $G_1^L, \ldots, G_n^L$, and unfold G^R (not necessarily completely) obtaining unfold tree goals $G_1^R, \ldots, G_m^R$. This is depicted in Figure 9. Then the proof holds if

$$G_1^L \vee \ldots \vee G_n^L \models G_1^R \vee \ldots \vee G_m^R$$

or alternatively, if $G_i^L \models G_1^R \vee \ldots \vee G_m^R$ for all $1 \leq i \leq n$. The justification for this result comes from the logical semantics of unfold: we have that $G^L \rightarrow G_1^L \vee \ldots \vee G_n^L$, and $G_j^R \rightarrow G^R$ for all j such that $1 \leq j \leq m$. By a chain of implications we may conclude $G^L \models G^R$.

More specifically, but with some loss of generality, the proof holds if

$$\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : G_i^L \models G_j^R.$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $G_i^L \models G_j^R$.

Note that since we replace the global satisfaction criterion by local criteria, our proof method is therefore incomplete in cases where we need to perform some unfolds of G^R, that is, when proving relative safety assertions. Unfold of G^R is not needed for proving traditional safety assertions.

Our proof method can also be viewed as checking that the set of states represented by the symbolic formula G^R is reachable, whenever the set G^L is reachable. This is done by showing that a frontier of states that reach G^L also reaches G^R. If G^L is to be reachable from the initial state, it must be through at least one of the states in this frontier. And since from all states in the frontier G^R is reachable, G^R must also be reachable from the initial state.

5.1 Proof Rules

We now present a calculus for proving relative safety assertions. To handle the possibly infinite unfoldings of G^L and G^R (see Figure 9), we shall depend on the use *coinduction* for the unfolding of G^L.

Proof by coinduction proceeds by assuming everything we like as long as we do not violate any facts. While assuming a set of assertions of the form $G^L \models G^R$ collected along an unfold path, we prove another assertion on the path, making it unnecessary to unfold the path further. For the use of coinduction, we now give the following definition.

Definition 6 (Proof Obligation). *A* proof obligation *is of the form* $\tilde{A} \vdash G^L \models G^R$*, where* G^L *and* G^R *are goals and* $\tilde{A}$ *is a set of assertions that are* assumed.

The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$ contains assertions that can be used coinductively to discard the proof obligation at hand.

Our proof rules are presented in Figure 10. Each rule operates on the (possibly empty) set of proof obligations Π, by selecting a proof obligation from Π and attempting to discard it. In this process, new proof obligations may be produced. The proof process is typically centered around unfolding the goals in proof obligations.

The *left unfold and coinduction* (LU+C) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π, is added as an assumption to every newly produced proof obligation, opening the door to using coinduction in the proof.

(LU+C) $$\frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{G^L \models G^R\} \vdash G_i^L \models G^R\}} \quad \text{unfold}(G^L) = \{G_1^L, \ldots, G_n^L\}$$

(RU) $$\frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \{\tilde{A} \vdash G^L \models G_i^R\}} \quad G_i^R \in \text{unfold}(G^R)$$

(AP) $$\frac{\Pi \cup \{\tilde{A} \vdash G^L, \phi \models G^R\}}{\Pi \cup \{\tilde{A} \vdash G_1^R\theta, \phi \models G^R\}} \quad \begin{array}{l} G_1^L \models G_1^R \in \tilde{A} \text{ and there} \\ \text{exists a renaming } \theta \text{ s.t.} \\ G^L \models G_1^L\theta \end{array}$$

(DP) $$\frac{\Pi \cup \{\tilde{A} \vdash G^L, \phi \models G^R\}}{\Pi} \quad \begin{array}{l} \text{There exists a renaming} \\ \theta \text{ s.t. } G^L \models G^R\theta \end{array}$$

(SPL) $$\frac{\Pi \cup \{\tilde{A} \vdash G^L \models G^R\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash G^L, \phi_i \models G^R\}} \quad \phi_1 \vee \ldots \vee \phi_k \text{ is true.}$$

Fig. 10. Proof Rules

Example 7 (Proving Symmetry). We exemplify our proof rules using a proof of a symmetry property of the 2-process bakery algorithm (Figure 2):

$$p([P_1, P_2], T_1, T_2) \models p([P_2, P_1], T_2, T_1). \quad (1)$$

Initially, $\Pi = \{\emptyset \vdash p([P_1, P_2], T_1, T_2) \models p([P_2, P_1], T_2, T_1)\}$.

Using the rule LU+C, and all the CLP rules of Figure 2, we perform a left unfold of $G^L = p([P_1, P_2], T_1, T_2)$, obtaining a new set of proof obligations Π'. In particular, by the unfold of CLP rule `r1`, Π' includes the obligation (O1):

$$\tilde{A}' \vdash p([P_1', P_2], T_1', T_2), P_1 = 1, P_1' = 0, T_1 = T_2 + 1 \models p([P_2, P_1], T_2, T_1),$$

where $\tilde{A}' = \{p([P_1, P_2], T_1, T_2) \models p([P_2, P_1], T_2, T_1)\}$.

By the unfold of CLP rule `init`, Π' also includes the obligation (O2):

$$\tilde{A}' \vdash P_1 = P_2 = 0, T_1 = T_2 = 0 \models p([P_2, P_1], T_2, T_1).$$

Other than these two obligations, Π' also includes the result of unfolding using the rules `e1`, `x1`, `r2`, `e2`, and `x2`.

The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. Note that only one unfolded goal is used. Now, in practice, it is generally not known which reduct G_i^R of G^R is the one we need later, or indeed if G^R itself is needed later.

Returning to our example, by unfolding $G^R = p([P_2, P_1], T_2, T_1)$ of (O1) using proof rule RU and CLP rule `r2` of Figure 2, we obtain Π'' which includes (O3):

$$\begin{array}{l}\tilde{A}' \vdash p([P_1', P_2], T_1', T_2), P_1 = 1, P_1' = 0, T_1 = T_2 + 1 \models \\ \quad p([P_2, P_1''], T_2, T_1''), P_1 = 1, P_1'' = 0, T_1 = T_2 + 1\end{array}$$

Similarly, by unfolding the rhs of (O2) using RU and the CLP rule `init`, we obtain Π''' which includes the obligation (O4):

$$\tilde{A}' \vdash P_1 = P_2 = 0, T_1 = T_2 = 0 \models P_1 = P_2 = 0, T_1 = T_2 = 0.$$

The rule *assumption proof* (AP) transforms an obligation by using an assumption, and realizes the coinduction principle (since assumptions can only be created by the rule (LU+C)).

Continuing our example, we can now immediately prove (O3) by rule AP, and applying the original symmetry assertion (1) which is included in the set of assumed assertions $\tilde{A}'$ of (O3). More concretely, we apply (1) to the lhs of (O3) obtaining the goal $p([P_2, P_1'], T_2, T_1'), P_1 = 1, P_1' = 0, T_1 = T_2 + 1$, which clearly implies the rhs of (O3) by renaming of each double primed variables to its single primed version.

The rule *direct proof* (DP) discards a proof obligation when it can be directly proven that it holds, possibly by some renaming of variables. This rule is used to discharge (O4), since it is immediately clear that it holds. The renaming θ that we apply here is the identity.

Finally, the rule *split* (SPL) converts a proof obligation into several, more specialized ones.

Given an assertion $G^L \models G^R$, a proof shall start with $\Pi = \{\tilde{A} \vdash G^L \models G^R\}$, and proceed by repeatedly applying the rules in Figure 10 to it. The conditions in which a proof can be completed are stated in the following theorem.

Theorem 1 (Proof of Assertions). *A safety assertion $G^L \models G^R$ holds if, starting with the proof obligation $\Pi = \{\emptyset \vdash G^L \models G^R\}$, there exists a sequence of applications of proof rules that results in $\Pi = \emptyset$. The safety assertion holds conditionally on $\tilde{A}$ if we start with $\Pi = \{\tilde{A} \vdash G^L \models G^R\}$, where $\tilde{A} \neq \emptyset$.*

Our proof method can be used to prove traditional safety assertion $G^L \models \Psi$, to prove relative safety assertion $G^L \models G^R$, where G^R contains an atom, and to prove traditional safety assertion using other assertions, e.g., relative safety assertions representing symmetry, possibly obtaining smaller proof. For the last use we start a prove of traditional safety assertion with a non-empty set of assumed assertions.

The proof rules above are sufficient in principle for our purposes. However, there is a very important principle which gives rise to an optimization: *redundancy* between obligations which essential idea is based on the observation that in proving $G^L \models G^R$, we may obtain a goal G^L_i by a sequence of unfolds from G^L, and prove the obligation $G^L_i \models G^R$. Using this we can try to establish $G^L_j, \phi \models G^R$ in another part of the tree, where $i \neq j$, where there exists a renaming θ such that $G^L_j \models G^L_i\theta$. Here, we *reuse* the proof of $G^L_i \models G^R$ in the proof of $G^L_j, \phi \models G^R$.

A fundamental question in proving relative safety assertion $G^L \models G^R$ in general, is how to interleave the unfolding of the lhs versus the rhs. For this we can repeatedly apply left-unfolding on G^L either until "looping", that is, until each path in the tree contains a repeated occurrence of a program counter, or the final goal of the path is a constraint. This is because coinduction is likely to be applicable at a looping point.

6 Implementation and Experiments

We implemented our proof algorithm as regular CLP($\mathcal{R}$) [15] programs. Our prototype implementations use coinduction, and a tabling mechanism for storing assumed assertions. We run our prototypes using a 2 GHz Pentium 4 Xeon machine with 2 GB of RAM.

Our first prototype is for proving relative assertions. Here we hope that the symmetry proof using coinduction concludes in just 1 level of unfold of both lhs and rhs of the assertion, because this is the case for *perfectly symmetric* programs. These include bakery algorithm and dining philosophers' problem. In these examples, every transition from state s to t has its symmetric counterpart that maps $\pi(s)$ to the $\pi(t)$, where π an automorphism of states. Our implementation therefore first tries to check goals obtained from 1 level of both lhs and rhs unfold. For each goal in the lhs frontier, it tries to search for a goal in the rhs of depth 1, such that the original symmetry assertion is applicable coinductively. Where the proof does not conclude in this manner, we have a program with imperfect symmetry, such is the case with the simple priority mutual exclusion and Szymanski's algorithm. In this case, general depth-first traversal of lhs subtree is initiated. For producer-consumer problem, we do not perform any lhs unfolding.

Experimental results in proving relative safety assertions are shown in Table 1, where A#=number of verified assertions, LSt=number of visited lhs goals, RSt=number of visited rhs goals, and T=time in seconds. In *ProblemName-N*, *N* denotes the number of processes, except for *Prod/Cons-N* where *N* denotes that there are *N* produce and consume operations. Note that we could not complete the experiment for 6-process bakery algorithm and 3-process Szymanski's algorithm after a few hours.

We also implemented a second prototype to prove safety assertions of the form $G \models$ *false* with or without assumed relative safety assertions (e.g., symmetry). $G \models$ *false* declares non-reachability of error states G.

A coinductive verification requires matching between the goal in an assertion and an assumed assertion such that the said assertion can be proven coinductively. As is common in the literature, for verification using symmetry we need to define a set of *canonical representatives* of the equivalence class of goals induced by given symmetry, such that the matching can be done efficiently among representatives. Unfortunately, finding all the canonical representatives of a goal is a hard problem known as the *orbit*

Table 1. Relative Safety Proof Experimental Results

Problem	A#	LSt	RSt	T
Bakery-2	1	9	27	0.00
Bakery-3	2	44	254	0.10
Bakery-4	3	147	1557	11.28
Bakery-5	4	424	7804	2320.3
Bakery-6	5	∞	∞	∞
Philosopher-3	1	19	124	0.01

Problem	A#	LSt	RSt	T
Philosopher-4	1	24	232	0.02
Priority	1	43	220	0.04
Szymanski-2	8	362	28419	59.11
Szymanski-3	16	∞	∞	∞
Prod/Cons-10	2	0	170	0.19
Prod/Cons-20	2	0	530	1.88

problem [2]. Our solution here is to try to generate canonical representatives of a goal only up to a constant number, and we employ a sorting algorithm as our canonicalization function. We note, however, that canonicalization is not hard for dining philosophers' problem since for this problem it is a cyclic shift which is linear to the permutable domain size (cf. [2]). Also that neither sorting nor cyclic shift is necessary when using serializability assertions.

The results are shown in Table 2 (a). The proof of traditional safety does not require right unfolding, hence there is no column for RSt value. We ran the bakery, Peterson's, Lamport's fast mutual exclusion and Szymanski's algorithms proving mutual exclusion. Note that we do not prove the symmetry assertions of some of the problems (e.g., Szymanski-3). For the dining philosophers' problem, we prove that there cannot be

Table 2. Safety Proof Experimental Results

Problem	CLP/Coinductive Tabling				Delzanno-Podelski
	No Assertion		W/ Assertion		
	LSt	T	LSt	T	# Facts
Bakery-2	15	0.00	8	0.00	13
Bakery-3	296	0.07	45	0.01	109
Bakery-4	4624	6.60	191	0.20	963
Bakery-5	∞	∞	677	2.88	
Bakery-6	∞	∞	2569	49.08	
Bakery-7	∞	∞	11865	1052.32	
Peterson-2	105	0.05	10	0.00	
Peterson-3	20285	119.03	175	0.15	
Peterson-4	∞	∞	3510	11.98	
Lamport-2	143	0.02	72	0.02	
Lamport-3	4255	1.13	707	0.40	
Lamport-4	∞	∞	5626	7.63	
Szymanski-2	240	0.08	84	0.02	
Szymanski-3	10883	35.43	3176	2.91	
Philosopher-3	882	0.51	553	0.30	
Philosopher-4	4293	27.77	2783	9.67	
Prod/Cons-10	664	0.10	171	0.02	
Prod/Cons-20	2314	1.90	331	0.04	

(a) Stored Assertions and Time

Problem Type	% Reduction	
	LSt	T
Bakery	76%	78%
Peterson	95%	99.9%
Lamport	67%	65%
Szymanski	68%	83%
Philosopher	36%	53%
Prod/Cons	87%	94%

(b) % Reduction

more than $N/2$ philosophers simultaneously eating. For the producer-consumer problem, each $\texttt{pro}_i\texttt{()}$ increments a variable x, and $\texttt{con}_j\texttt{()}$ decrements it. Here we verify that the value of x can never be more than $2n$.

Bakery algorithm has infinite reachable states, and therefore cannot be handled by finite-state model checkers. We compare our search space the results of the CLP-based system of Delzanno and Podelski [4]. As also noted by Delzanno and Podelski, the problem does not scale well to larger number of processes, but using symmetry, we have pushed its verification limit to 7 processes without abstraction.

In Table 2 (b) we summarize the effectiveness of the use of a variety of relative safety assertions. The use of symmetry assertion effectively reduces the search space of perfectly symmetric problems (bakery, Peterson's, Lamport's fast mutex, dining philosophers). However, the reduction for Szymanski's algorithm is competitive with perfectly symmetric problems, showing that "not-quite" symmetry reduction is worth pursuing. The use of rotational symmetry in the dining philosophers' problem is, expectedly, less effective. We also note that we managed to obtain a substantial reduction of state space for the producer/consumer problem. Reduction in time roughly corresponds to those of state space.

Finally, comparing Table 1 and 2, the proof of relative safety assertions are no easier than the proof of traditional safety assertions, even with coinduction. This is because of the need to perform rhs unfold when proving relative safety.

7 Conclusion

In this paper, we introduced a novel assertion called relative safety. This can be uniquely used to assert structural properties of programs. We chose a driving application area of symmetry, and demonstrated that, by using relative safety, we could accommodate a larger class of programs than have been previously considered by other means. We provided a proof system, based upon well understood computational steps of unfolding, and introduced a new coinductive tabling mechanism. We then ran some experiments in order to show the practical potential of our algorithm. Further work is to discover more important classes of structural properties for which relative safety can be used.

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM TOPLAS*, 16(5):1543–1571, September 1994.
2. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *10th CAV*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998.
3. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *5th CAV*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.
4. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. STTT*, 3(3):250–270, 2001.
5. X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*, pages 175–184. IEEE Computer Society Press, 2000.

6. E. A. Emerson. From asymmetry to full symmetry: New techniques for symmetry reductions in model checking. In L. Pierre and T. Kropf, editors, *10th CHARME*, volume 1703 of *LNCS*, pages 142–156. Springer, 1999.
7. E. A. Emerson, J. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *15th LICS*, pages 121–131. IEEE Computer Society Press, 2000.
8. E. A. Emerson and A. P. Sistla. Model checking and symmetry. In *5th CAV*, volume 697 of *LNCS*, pages 463–478. Springer, 1993.
9. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions. *ACM TOPLAS*, 19(4):617–638, July 1997.
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In M. Leuschel, A. Podelski, C. R. Ramakrishnan, and U. Ultes-Nitsche, editors, *2nd VCL*, pages 85–96, 2001.
11. L. Fribourg. Constraint logic programming applied to model checking. In *9th LOPSTR*, volume 1817 of *LNCS*, pages 30–41. Springer, 1999.
12. G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *18th RTSS*, pages 230–239. IEEE Computer Society Press, 1997.
13. C. N. Ip and D. L. Dill. Better verification through symmetry. *FMSD*, 9(1/2):41–75, 1996.
14. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
15. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
16. J. Jaffar, A. Santosa, and R. Voicu. A CLP proof method for timed automata. In *25th RTSS*, pages 175–186. IEEE Computer Society Press, 2004.
17. M. Leuschel and T. Massart. Infinite-state model checking by abstract interpretation and program specialization. In *9th LOPSTR*, volume 1817 of *LNCS*, pages 62–81. Springer, 1999.
18. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. La u, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *1st CL*, volume 1861 of *LNCS*, pages 384–398. Springer, 2000.
19. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *9th CAV*, volume 1254 of *LNCS*, pages 143–154. Springer, 1997.
20. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM TOPLAS*, 26(4):702–734, July 2004.
21. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM TOSEM*, 9(2):133–166, April 2000.
22. F. Wang. Efficient data structure for fully symbolic verification of real-time systems. In S. Graf and M. I. Schwartzbach, editors, *6th TACAS*, volume 1785 of *LNCS*, pages 157–171. Springer, 2000.
23. H. Weyl. *Symmetry*. Princeton University Press, 1952.

Resource Usage Analysis for the π-Calculus

Naoki Kobayashi[1], Kohei Suenaga[2], and Lucian Wischik[3]

[1] Tohoku University
koba@ecei.tohoku.ac.jp
[2] University of Tokyo
kohei@yl.is.s.u-tokyo.ac.jp
[3] Microsoft Corporation
lwischik@microsoft.com

Abstract. We propose a type-based resource usage analysis for the π-calculus extended with resource creation/access primitives. The goal of the resource usage analysis is to statically check that a program accesses resources such as files and memory in a valid manner. Our type system is an extension of previous behavioral type systems for the pi-calculus, and can guarantee the safety property that no invalid access is performed, as well as the property that necessary accesses (such as the close operation for a file) are eventually performed unless the program diverges. A sound type inference algorithm for the type system is also developed to free the programmer from the burden of writing complex type annotations. Based on the algorithm, we have implemented a prototype resource usage analyzer for the π-calculus. To the authors' knowledge, ours is the first type-based resource usage analysis that deals with an expressive concurrent language like the π-calculus.

1 Introduction

Computer programs access many external resources, such as files, library functions, device drivers, etc. Such resources are often associated with certain access protocols; for example, an opened file should be eventually closed and after the file has been closed, no read/write access is allowed. The aim of resource usage analysis [9] is to statically check that programs conform to such access protocols. Although a number of approaches, including type systems and model checking, have been proposed so far for the resource usage analysis or similar analyses [1, 5–7, 9], most of them focused on analysis of sequential programs, and did not treat concurrent programs, especially those involving dynamic creation/passing of channels and resources.

In the present paper, we propose a type-based method of resource usage analysis for *concurrent languages*. Dealing with concurrency is especially important because concurrent programs are hard to debug, and also because actual programs accessing resources are often concurrent. We use the π-calculus (extended with resource primitives) as a target language so that our analysis can be applied to a wide range of concurrency primitives (including those for dynamically creating and passing channels) in a uniform manner.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 298–312, 2006.
© Springer-Verlag Berlin Heidelberg 2006

For the purpose of analyzing resource usage, we extend previous behavioral type systems for the π-calculus [3, 8]. The idea of the behavioral types [3, 8] is to use CCS-like processes as types. The types express abstract behavior of processes, so that certain properties of processes can be verified by verifying the corresponding properties of their types, using, for example, model checking techniques. The latter properties (of CCS-like types) are more amenable to automatic verification techniques like model checking than the former ones, because the types do not have channel mobility and also because the types typically represent only the behavior of a part of the entire process.

Following the previous behavioral types, we use CCS-like types to express resource-wise access behaviors of a process and construct a type system which guarantees that any well-typed process uses resources in a valid manner. The main contributions of the present paper are:

- Adaption of behavioral types (for pure π-calculus) [3, 8] to the π-calculus extended with resource access primitives.
- Realization of fully automatic verification (while making the analysis more precise than [8]). Igarashi and Kobayashi [8] gave only an abstract type system, without giving a concrete type inference algorithm. Chaki et al. [3] requires type annotations. The full automation was enabled by a combination of a number of small ideas, like inclusion of hiding and renaming as type constructors (Igarashi and Kobayashi [8] used a fragment without hiding and renaming, and Chaki et al. [3] used a fragment without renaming), approximation of a CCS-like type by a Petri net (to reduce the problem of checking conformance of inferred types to resource usage specification).
- Verification of not only the usual safety property that an invalid resource access does not occur, but also an extended safety (which we call *partial liveness*) that necessary resource accesses (e.g. closing of a file) are eventually performed unless the whole process diverges. The partial liveness is not guaranteed by Chaki et al.'s type system [3]. A noteworthy point about our type system for guaranteeing the partial liveness is that it is parameterized by a mechanism that guarantees deadlock-freedom (in the sense of Kobayashi's definition [13]). So, our type system can be combined with *any* mechanism (model checking, abstract interpretation, another type system, or whatever) to verify deadlock-freedom.
- Implementation of a prototype resource usage analyzer based on the proposed method. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

The rest of this paper is structured as follows. Section 2 introduces an extension of the π-calculus with primitives for creating and accessing resources. Section 3 introduces a type system for resource usage analysis. Section 4 gives a type inference algorithm for the type system. Section 5 presents our prototypical implementation. Section 6 discusses related work. Section 7 concludes. For lack of space, proofs and some explanations have been omitted. They are found in the full version of this paper [15].

2 Language

Let x, y, z range over a countably infinite set **Var** of variables, let values v range over variables and also the two constant values **true** and **false**, let tags t range over $\{\emptyset, \mathbf{c}\}$, let ξ range over a set of *access labels*, and let Φ (called a *trace set*) denote a set of sequences of access labels, possibly ending with a special label $\downarrow$, that is closed under the prefix operation. We write $\widetilde{x}$ for a sequence $x_1, \ldots, x_n$ of variables, and similarly $\widetilde{v}$, and define $\Phi^{-\xi} = \{s \mid \xi s \in \Phi\}$. Let L range over *reduction labels* $\{x^\xi \mid x \in \mathbf{Var}\} \cup \{\tau\}$.

$$P ::= \mathbf{0} \mid (P \,|\, Q) \mid \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q \mid (\nu x)\, P \mid *P$$
$$\mid \overline{x}_t\langle\widetilde{v}\rangle.\, P \mid x_t(\widetilde{y}).\, P \mid (\mathfrak{N}^\Phi x)P \mid \mathbf{acc}_\xi(x).P$$

Structural preorder $\preceq$ is as follows. $P \equiv Q$ stands for $(P \preceq Q) \wedge (Q \preceq P)$.

$$P \,|\, \mathbf{0} \equiv P \qquad P \,|\, Q \equiv Q \,|\, P \qquad P \,|\, (Q \,|\, R) \equiv (P \,|\, Q) \,|\, R \qquad *P \preceq *P \,|\, P$$

$$(\nu x)\, P \,|\, Q \preceq (\nu x)\, (P \,|\, Q) \text{ and } (\mathfrak{N}^\Phi x)P \,|\, Q \preceq (\mathfrak{N}^\Phi x)(P \,|\, Q) \textit{ if } x \textit{ not free in } Q$$

$$\frac{P \preceq P' \quad Q \preceq Q'}{P \,|\, Q \preceq P' \,|\, Q'} \qquad \frac{P \preceq Q}{(\nu x)\, P \preceq (\nu x)\, Q} \qquad \frac{P \preceq Q}{(\mathfrak{N}^\Phi x)P \preceq (\mathfrak{N}^\Phi x)Q}$$

Labeled relation $\xrightarrow{L}$ is as follows. Write $P \longrightarrow Q$ when $P \xrightarrow{L} Q$ for some L, and $\longrightarrow^*$ for reflexive and transitive closure of $\longrightarrow$. Define $\mathit{target}(x^\xi) = \{x\}$ and $\mathit{target}(\tau) = \emptyset$.

$$\overline{x}_t\langle\widetilde{z}\rangle.\, P \,|\, x_{t'}(\widetilde{y}).\, Q \xrightarrow{\tau} P \,|\, [\widetilde{z}/\widetilde{y}]Q \qquad \mathbf{acc}_\xi(x).P \xrightarrow{x^\xi} P \qquad \begin{array}{l}\mathbf{if\ true\ then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} P \\ \mathbf{if\ false\ then}\ P\ \mathbf{else}\ Q \xrightarrow{\tau} Q\end{array}$$

$$\frac{P \xrightarrow{L} Q}{P \,|\, R \xrightarrow{L} Q \,|\, R} \qquad \frac{P \xrightarrow{L} Q \quad x \notin \mathit{target}(L)}{(\nu x)\, P \xrightarrow{L} (\nu x)\, Q} \qquad \frac{P \xrightarrow{x^\xi} Q}{(\mathfrak{N}^\Phi x)P \xrightarrow{\tau} (\mathfrak{N}^{\Phi^{-\xi}} x)Q}$$

$$\frac{P \xrightarrow{L} Q \quad x \notin \mathit{target}(L)}{(\mathfrak{N}^\Phi x)P \xrightarrow{L} (\mathfrak{N}^\Phi x)Q} \qquad \frac{P \preceq P' \quad P' \xrightarrow{L} Q' \quad Q' \preceq Q}{P \xrightarrow{L} Q}$$

Fig. 1. Process language

The process language P is in Figure 1. The first line is standard π-calculus – $(\nu x)\, P$ declares a new channel with bound name x, $*P$ is replication, and parallel composition $|$ binds less tightly than the prefixes. We often omit trailing $\mathbf{0}$. Bound and free variables are defined as normal. We identify processes up to α-conversion, and so assume that bound variables are always different from each other and from free variables.

The input command $x_t(\widetilde{y}).\, P$ waits for input on channel x with bound formal parameters $\widetilde{y}$ and then behaves as P. The output command $\overline{x}_t\langle\widetilde{v}\rangle.\, P$ sends $\widetilde{v}$ along x and then behaves as P. The attribute t is either $\mathbf{c}$ (indicating that if the input is executed then it will succeed unless the whole process diverges) or $\emptyset$ (which does not give the guarantee). We often omit $\emptyset$. Note that the attributes do not affect the operational semantics of processes. Typically the attributes

have been inferred by some deadlock analysis tool such as `TyPiCal` [10–12, 14]. For this paper, we assume that the correctness of the attributes are ensured by whichever deadlock-analysis tool used to make the annotations.

For the final line in the definition of processes, $(\mathfrak{N}^{\Phi} x)P$ declares a resource with bound name x which is to be accessed according to specification Φ, and $\mathbf{acc}_{\xi}(x).P$ performs access ξ on resource x and then behaves like P. Resources here are an abstraction of real-world resources such as files or objects. In this paper we consider accesses such as I for initialize, R for read, W for write and C for close. For example, $(\mathfrak{N}^{(I(R+W)^* C \downarrow)^{\#}} x)P$ creates a resource that should be first initialized, read or written an arbitrary number of times, and then closed. The symbol $\downarrow$ at the end indicates that the final close is required eventually to occur. Here, $(S)^{\#}$ is the prefix closure of S, i.e., $\{s \mid ss' \in S\}$. We write ϵ for the empty access sequence. We write $\mathbf{init}(x).P$ for $\mathbf{acc}_I(x).P$, and similarly $\mathbf{read}(x)$, $\mathbf{write}(x)$, $\mathbf{close}(x)$. We do not fix the syntax of Φ. Our type system is independent of the choice of the language for describing the specificaiton Φ (except for the sub-algorithm for type-checking discussed in Section 4.1, where we assume that Φ is a regular language).

We treat resources as primitives in this paper, and give operational semantics where $\mathbf{acc}_{\xi}(x).P$ is non-blocking. This is for simplicity. It would also be possible to treat a resource with (say) three access labels as a tuple of three channels. This would allow previous work [3, 8] to infer some of the properties of this paper, albeit with less precision and more complexity. Also in this paper we have specifications Φ apply only to a single resource. To model a program with two co-declared resources as in [8] with intertwined specifications, we would instead merge them into a single resource with a single specification.

The operational semantics of the language are given in Figure 1, through a *structural preorder* $\preceq$ and a labeled reduction relation $\xrightarrow{L}$. Notice that invalid resource access sets $\Phi = \emptyset$, valid access removes a prefix from Φ, and complete access results in $\Phi = \{\epsilon, \downarrow\}$).

$$(\mathfrak{N}^{(IC\downarrow)^{\#}} x)\mathbf{read}(x).\mathbf{0} \to (\mathfrak{N}^{\emptyset} x)\mathbf{0} \qquad \text{(invalid access)}$$

$$(\mathfrak{N}^{(IC\downarrow)^{\#}} x)\mathbf{init}(x).\mathbf{0} \to (\mathfrak{N}^{(C\downarrow)^{\#}} x)\mathbf{0} \qquad \text{(valid access)}$$

$$(\mathfrak{N}^{(IC\downarrow)^{\#}} x)\mathbf{init}(x).\mathbf{close}(x).\mathbf{0} \to (\mathfrak{N}^{\{\epsilon,\downarrow\}} x)\mathbf{0} \qquad \text{(complete access)}$$

We are concerned with the following properties.

Definition 1. *1. A process P is* safe *if it does not contain a sub-expression of the form $(\mathfrak{N}^{\emptyset} x)Q$.*
2. A process P is partially live *if $\downarrow \in \Phi$ whenever $P \longrightarrow^* \preceq (\widetilde{\nu \mathfrak{N}})(\mathfrak{N}^{\Phi} x)Q \not\longrightarrow$.*

The first property means that the process has not performed any invalid access. The second property means that necessary accesses are eventually performed before the whole process converges. In the next section, we shall develop a type system that guarantees the safety and partial liveness.

Example 1. The following example process is safe and partially live. It uses internal synchronization to ensure that the resource x is accessed in a valid order.

$$
\begin{array}{ll}
(\mathfrak{N}^{(IR^*C)^\#}x)(\nu y)\,(\nu z)\,\big(& \\
\quad \mathbf{init}(x).(\overline{y}\langle\,\rangle\,|\,\overline{y}\langle\,\rangle) & \textit{/* initialize x, and send signals */} \\
\quad |\, y_{\mathbf{c}}(\,).\,\mathbf{read}(x).\overline{z}\langle\,\rangle & \textit{/* wait on y, then read x, and signal on z */} \\
\quad |\, y_{\mathbf{c}}(\,).\,\mathbf{read}(x).\overline{z}\langle\,\rangle & \textit{/* wait on y, then read x, and signal on z */} \\
\quad |\, z_{\mathbf{c}}(\,).\,z_{\mathbf{c}}(\,).\,\mathbf{close}(x)\,\big) & \textit{/* wait on z, then close x */}
\end{array}
$$

3 Type System

3.1 Types

We first introduce the syntax of types. We use two categories of types: value types and behavioral types. The latter describes how a process accesses resources and communicates through channels. As mentioned in Section 1, we use CCS processes for behavioral types.

Definition 2 (types). *The sets of* value types σ *and* behavioral types A *are defined by:*

$$
\begin{array}{l}
\sigma ::= \textbf{\textit{bool}} \mid \textbf{\textit{res}} \mid \textbf{\textit{chan}}\langle (x_1:\sigma_1,\ldots,x_n:\sigma_n)A\rangle \\
A ::= \textbf{\textit{0}} \mid \alpha \mid a_t.A \mid x^\xi.A \mid \tau_t.A \mid (A_1 \mid A_2) \mid A_1 \oplus A_2 \mid *A \\
\qquad \mid \langle y_1/x_1,\ldots,y_n/x_n\rangle A \mid (\nu x)\,A \mid \mu\alpha.A \mid A{\uparrow_S} \mid A{\downarrow_S} \\
a \textit{ (communication labels) } ::= x \mid \overline{x}
\end{array}
$$

A behavioral type A, which is a CCS process, describes what kind of communication and resource access a process may perform. $\mathbf{0}$ describes a process that performs no communication or resource access. The types $x_t.\,A$, $\overline{x}_t.\,A$, $x^\xi.A$ and $\tau_t.A$ describes process that first perform an action and then behave according to A; the actions are, respectively, an input on x, an output on x, an access operation ξ on x, and the invisible action. Attributes t denote whether an action is guaranteed to succeed. $A_1 \,|\, A_2$ describes a process that performs communications and resource access according to A_1 and A_2, possibly in parallel. $A_1 \oplus A_2$ describes a process that behaves according to either A_1 or A_2. $*A$ describes a process that behaves like A an arbitrary number of times, possibly in parallel. $\langle y_1/x_1,\ldots,y_n/x_n\rangle A$, abbreviated to $\langle\widetilde{y}/\widetilde{x}\rangle A$, denotes simultaneous renaming of $\widetilde{x}$ with $\widetilde{y}$ in A. $(\nu x)\,A$ describes a process that behaves like A for some hidden channel x. For example, $(\nu x)\,(x.\,\overline{y}\,|\,\overline{x})$ describes a process that performs an output on y after the invisible action on x. The type $\mu\alpha.A$ describes a process that behaves like a recursive process defined by $\alpha \triangleq A$. The type $A{\uparrow_S}$ describes a process that behaves like A, except that actions whose targets are in S are replaced by the invisible action τ, while $A{\downarrow_S}$ describes a process that behaves like A, except that actions whose targets are not in S are replaced by τ. The formal semantics of behavioral types is defined later using labeled transition semantics.

As for value types, **bool** is the type of booleans. **res** is the type of resources. The type $\mathbf{chan}\langle(x_1:\sigma_1,\ldots,x_n:\sigma_n)A\rangle$, abbreviated to $\mathbf{chan}\langle(\widetilde{x}:\widetilde{\sigma})A\rangle$, describes channels carrying tuples consisting of values of types $\sigma_1,\ldots,\sigma_n$. Here the type A approximates how a receiver on the channel may use the elements

$x_1, \ldots, x_n$ of each tuple for communications and resource access. For example, $\mathbf{chan}\langle(x:\mathbf{res}, y:\mathbf{res})x^R.y^C\rangle$ describes channels carrying a pair of resources, where a party who receives the actual pair (x', y') will first read x' and then close y'. We sometimes omit $\widetilde{\sigma}$ and write $\mathbf{chan}\langle(\widetilde{x})A\rangle$ for $\mathbf{chan}\langle(\widetilde{x}:\widetilde{\sigma})A\rangle$. When $\widetilde{x}$ is empty, we also write $\mathbf{chan}\langle\rangle$.

Note that $\langle\widetilde{y}/\widetilde{x}\rangle$ is treated as a constructor rather than an operator for performing the actual substitution. We write $[\widetilde{y}/\widetilde{x}]$ for the latter throughout this paper. $\langle\widetilde{y}/\widetilde{x}\rangle A$ is slightly different from the *relabeling* of the standard CCS [17]: $\langle y/x\rangle(x \,|\, \overline{y})$ allows the communication on y, but the relabeling of CCS does not. This difference calls for the introduction of a special transition label $\{x, \overline{y}\}$ in Section 3.2.

$(\nu x)\, A$, $\langle\widetilde{y}/\widetilde{x}\rangle A$, and $A{\uparrow_S}$ bind x, $\widetilde{x}$, and the variables in S respectively. We write $\mathbf{FV}(A)$ for the set of free variables in A. We identify behavioral types up to renaming of bound variables. In the rest of this paper, we require that every channel type $\mathbf{chan}\langle(x_1:\sigma_1, \ldots, x_n:\sigma_n)A\rangle$ must satisfy $\mathbf{FV}(A) \subseteq \{x_1, \ldots, x_n\}$. For example, $\mathbf{chan}\langle(x{:}\mathbf{res})x^R\rangle$ is a valid type but $\mathbf{chan}\langle(x{:}\mathbf{res})y^R\rangle$ is not. By abuse of notation, we write $\langle v_1/x_1, \ldots, v_n/x_n\rangle A$ for $\langle v_{i_1}/x_{i_1}, \ldots, v_{i_k}/x_{i_k}\rangle A$ where $\{v_{i_1}, \ldots, v_{i_k}\} = \{v_1, \ldots, v_n\}\backslash\{\mathbf{true}, \mathbf{false}\}$. For example, $\langle\mathbf{true}/x, y/z\rangle A$ stands for $\langle y/z\rangle A$.

3.2 Semantics of Behavioral Types

We give a labeled transition relation $\xrightarrow{l}$ for behavioral types. The transition labels l are

$$l \;::=\; x \mid \overline{x} \mid x^{\xi} \mid \tau \mid \{x, \overline{y}\}$$

The label $\{x, \overline{y}\}$ indicates the potential to react in the presence of a substitution that identifies x and y. We also extend *target* to the function on transition labels by:

$$\mathit{target}(x) = \mathit{target}(\overline{x}) = \{x\} \qquad \mathit{target}(\{x, \overline{y}\}) = \{x, y\}$$

Figure 2 shows a part of the definition of the transition relation $\xrightarrow{l}$ on behavioral types. For the complete definition, see the full paper [15]. We write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$. We also write $\xRightarrow{l}$ for $\Longrightarrow\xrightarrow{l}\Longrightarrow$.

$$a_t.A \xrightarrow{a} A \qquad x^{\xi}.A \xrightarrow{x^{\xi}} A \qquad \tau_t.A \xrightarrow{\tau} A$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A{\uparrow_S} \xrightarrow{\tau} A'{\uparrow_S}} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A{\uparrow_S} \xrightarrow{l} A'{\uparrow_S}}$$

$$\frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \subseteq S}{A{\downarrow_S} \xrightarrow{l} A'{\downarrow_S}} \qquad \frac{A \xrightarrow{l} A' \quad \mathit{target}(l) \cap S = \emptyset}{A{\downarrow_S} \xrightarrow{\tau} A'{\downarrow_S}}$$

Fig. 2. A Part of Definition of Transition semantics of behavioral types

Remark 1. $(\nu x)\,A$ should not be confused with $A\uparrow_{\{x\}}$. $(\nu x)\,A$ is the hiding operator of CCS, while $A\uparrow_{\{x\}}$ just replaces any actions on x with τ [8]. For example, $(\nu x)\,(x.\,y^\xi)$ cannot make any transition, but $(x.\,y^\xi)\uparrow_{\{x\}} \xrightarrow{\tau}\xrightarrow{y^\xi} \mathbf{0}\uparrow_{\{x\}}$.

We next define a predicate *disabled*(A, S) inductively as follows.

$$
\begin{array}{rl}
\mathit{disabled}(\mathbf{0}, S) & \\
\mathit{disabled}(x^\xi.A, S) & \text{if } \mathit{disabled}(A, S) \text{ and } x \notin S \\
\mathit{disabled}(a_{\mathbf{c}}.A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}(a_\emptyset.A, S) & \\
\mathit{disabled}(\tau_{\mathbf{c}}.A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}(\tau_\emptyset.A, S) & \\
\mathit{disabled}(A_1 \mid A_2, S) & \text{if } \mathit{disabled}(A_1, S) \text{ and } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(A_1 \oplus A_2, S) & \text{if } \mathit{disabled}(A_1, S) \text{ or } \mathit{disabled}(A_2, S) \\
\mathit{disabled}(*A, S) & \text{if } \mathit{disabled}(A, S) \\
\mathit{disabled}((\nu x)\,A, S) & \text{if } \mathit{disabled}(A, S\backslash\{x\}) \\
\mathit{disabled}(A\uparrow_{S'}, S) & \text{if } \mathit{disabled}(A, S\backslash S') \\
\mathit{disabled}(A\downarrow_{S'}, S) & \text{if } \mathit{disabled}(A, S \cap S') \\
\mathit{disabled}(\langle\widetilde{y}/\widetilde{x}\rangle A, S) & \text{if } \mathit{disabled}(A, \{z \mid [\widetilde{y}/\widetilde{x}]z \in S\}) \\
\mathit{disabled}(\mu\alpha.A, S) & \text{if } \mathit{disabled}([\mu\alpha.A/\alpha]A, S)
\end{array}
$$

Intuitively, *disabled*(A, S) means that A describes a process that may get blocked without accessing any resources in S.

The set $\mathbf{etraces}_x(A)$ defined below is the set of possible access sequences on x described by A.

Definition 3 (extended traces). *The set* $\boldsymbol{etraces}_x(A)$ *of extended traces is:*

$$
\begin{array}{l}
\{\xi_1 \cdots \xi_n \downarrow \mid \exists B.A\downarrow_{\{x\}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} B \wedge \mathit{disabled}(B, \{x\})\} \\
\cup\{\xi_1 \cdots \xi_n \mid \exists B.A\downarrow_{\{x\}} \stackrel{x^{\xi_1}}{\Longrightarrow} \cdots \stackrel{x^{\xi_n}}{\Longrightarrow} B\}
\end{array}
$$

We define the subtyping relation $A_1 \leq A_2$ below. Intuitively, $A_1 \leq A_2$ means that a process behaving according to A_1 can also be viewed as a process behaving according to A_2. To put in another way, $A_1 \leq A_2$ means that A_2 simulates A_1.[1] We define $\leq$ for only *closed* types, i.e., those not containing free type variables.

Definition 4 (subtyping). *The subtyping relation $\leq$ on closed behavioral types is the largest relation that satisfies the following properties:*

- *$A_1 \leq A_2$ and $A_1 \xrightarrow{l} A_1'$ implies $A_2 \stackrel{l}{\Longrightarrow} A_2'$ and $A_1' \leq A_2'$ for some A_2'.*
- *disabled(A_1, S) implies disabled(A_2, S) for any set S of variables.*

We often write $A_1 \geq A_2$ for $A_2 \leq A_1$, and write $A_1 \approx A_2$ for $A_1 \leq A_2 \wedge A_2 \leq A_1$.

[1] Note that the subtyping relation defined here is the converse of the one used in Igarashi and Kobayashi's generic type system [8].

3.3 Typing

We consider two kinds of judgments, $\Gamma \triangleright v : \sigma$ for values, and $\Gamma \triangleright P : A$ for processes. Γ is a mapping from a finite set of variables to value types. In $\Gamma \triangleright P : A$, the type environment Γ describes the types of the variables, and A describes the possible behaviors of P. For example, $x : \mathbf{chan}\langle (b : \mathbf{bool})\mathbf{0}\rangle \triangleright P : \overline{x} \mid \overline{x}$ implies that P may send booleans along the channel x twice. The judgment $y : \mathbf{chan}\langle (x : \mathbf{chan}\langle (b : \mathbf{bool})\mathbf{0}\rangle)\overline{x}\rangle \triangleright Q : y$ means that Q may perform an input on y once, and then it may send a boolean on the received value. Note that in the judgment $\Gamma \triangleright P : A$, the type A is an approximation of the behavior of P on free channels. P may do less than what is specified by A, but must not do more; for example, $x : \mathbf{chan}\langle (\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle : \overline{x} \mid \overline{x}$ holds but $x : \mathbf{chan}\langle (\,)\mathbf{0}\rangle \triangleright \overline{x}\langle\,\rangle.\, \overline{x}\langle\,\rangle : \overline{x}$ does not. Because of this invariant, if A does not perform any invalid access, neither does P.

We write $dom(\Gamma)$ for the domain of Γ. We write $\emptyset$ for the empty type environment, and write $x_1 : \tau_1, \ldots, x_n : \tau_n$ (where $x_1, \ldots, x_n$ are distinct from each other) for the type environment Γ such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. When $x \notin dom(\Gamma)$, we write $\Gamma, x : \tau$ for the type environment Δ such that $dom(\Delta) = dom(\Gamma) \cup \{x\}$, $\Delta(x) = \tau$, and $\Delta(y) = \Gamma(y)$ for $y \in dom(\Gamma)$. We define the *value judgment* relation $\Gamma \triangleright v{:}\sigma$ to be the least relation closed under

$$\Gamma, x{:}\sigma \triangleright x{:}\sigma \qquad \Gamma \triangleright \mathbf{true}{:}\mathbf{bool} \qquad \Gamma \triangleright \mathbf{false}{:}\mathbf{bool}.$$

We write $\Gamma \triangleright \widetilde{v}{:}\widetilde{\sigma}$ as an abbreviation for $(\Gamma \triangleright v_1{:}\sigma_1) \wedge \cdots \wedge (\Gamma \triangleright v_n{:}\sigma_n)$.

Figure 3 gives the rules for the relation $\Gamma \triangleright P : A$. We explain key rules below. In rule (T-OUT), the first premise $\Gamma \triangleright P : A_2$ implies that the continuation of the output process behaves like A_2, and the second premise $\Gamma \triangleright x : \mathbf{chan}\langle (\widetilde{y} : \widetilde{\sigma})A_1\rangle$

$$\frac{\Gamma \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle (\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad \Gamma \triangleright \widetilde{v} : \widetilde{\sigma}}{\Gamma \triangleright \overline{x}_t\langle \widetilde{v}\rangle.\, P : \overline{x}_t.\, (\langle \widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)} \quad \text{(T-OUT)}$$

$$\frac{\Gamma, \widetilde{y} : \widetilde{\sigma} \triangleright P : A_2 \quad \Gamma \triangleright x : \mathbf{chan}\langle (\widetilde{y} : \widetilde{\sigma})A_1\rangle \quad A_2{\downarrow}_{\{\widetilde{y}\}} \leq A_1}{\Gamma \triangleright x_t(\widetilde{y}).\, P : x_t.\, (A_2{\uparrow}_{\{\widetilde{y}\}})} \quad \text{(T-IN)}$$

$$\frac{\Gamma \triangleright P_1 : A_1 \qquad \Gamma \triangleright P_2 : A_2}{\Gamma \triangleright P_1 \mid P_2 : A_1 \mid A_2} \quad \text{(T-PAR)} \qquad\qquad \Gamma \triangleright \mathbf{0} : \mathbf{0} \quad \text{(T-ZERO)}$$

$$\frac{\Gamma \triangleright P : A}{\Gamma \triangleright *P : *A} \quad \text{(T-REP)} \qquad \frac{\Gamma \triangleright P : A \qquad \Gamma \triangleright x : \mathbf{res}}{\Gamma \triangleright \mathbf{acc}_\xi(x).P : x^\xi.A} \quad \text{(T-ACC)}$$

$$\frac{\Gamma \triangleright v : \mathbf{bool} \qquad \Gamma \triangleright P : A \qquad \Gamma \triangleright Q : A}{\Gamma \triangleright \mathbf{if}\ v\ \mathbf{then}\ P\ \mathbf{else}\ Q\ : A} \quad \text{(T-IF)} \qquad \frac{\Gamma, x : \mathbf{res} \triangleright P : A \qquad \mathbf{etraces}_x(A) \subseteq \Phi}{\Gamma \triangleright (\mathfrak{N}^\Phi x)P : A{\uparrow}_{\{x\}}} \quad \text{(T-NEWR)}$$

$$\frac{\Gamma, x : \mathbf{chan}\langle (\widetilde{y} : \widetilde{\sigma})A_1\rangle \triangleright P : A_2}{\Gamma \triangleright (\nu x)\, P : (\nu x)\, A_2} \quad \text{(T-NEW)} \qquad \frac{\Gamma \triangleright P : A' \qquad A' \leq A}{\Gamma \triangleright P : A} \quad \text{(T-SUB)}$$

Fig. 3. Typing Rules

implies that the tuple of values $\widetilde{v}$ being sent may be used by an input process according to $\langle\widetilde{v}/\widetilde{y}\rangle A_1$. Therefore, the whole behavior of the output process is described by $\overline{x}.\,(\langle\widetilde{v}/\widetilde{y}\rangle A_1 \mid A_2)$. Note that, as in previous behavioral type systems [3, 8], the resource access and communications made on $\widetilde{v}$ by the receiver of $\widetilde{v}$ are counted as the behavior of the output process. In rule (T-In), the first premise implies that the continuation of the input process behaves like A_2. Following previous behavioral type systems [3, 8], we split A_2 into two parts: $A_2{\downarrow_{\{\widetilde{y}\}}}$ and $A_2{\uparrow_{\{\widetilde{y}\}}}$. The first part describes the behavior on the received values $\widetilde{y}$ and is taken into account in the channel type. The second part describes the resource access and communications performed on other values, and is taken into account in the behavioral type of the input process. The condition $A_2{\downarrow_{\{\widetilde{y}\}}} \leq A_1$ requires that the access and communication behavior on $\widetilde{y}$ conforms to A_1, the channel arguments' behavior. In (T-New), the premise implies that P behaves like A, so that $(\nu x)\,P$ behaves like $(\nu x)\,A$. Here, we only require that x is a channel, unlike in the previous behavioral type systems for the π-calculus [8, 10]. That is because we are only interested in the resource access behavior; the communication behavior is used only for accurately inferring the resource access behavior. In (T-NewR), we check that the process's behavior A conforms to the resource usage specification Φ. Rule (T-Sub) allows the type A' of a process to be replaced by its approximation A.

Example 2. Consider the process $P = (\nu s)\,(*s(n,x,r).\,P_1 \;\mid\; (\mathfrak{N}^{\Phi} x)P_2)$, where:

$$P_1 = \textbf{if } n = 0 \textbf{ then } \overline{r}\langle\rangle \textbf{ else } (\nu r')\,(\overline{s}\langle n-1, x, r'\rangle \mid r'_{\mathbf{c}}().\,\textbf{read}(x).\overline{r}\langle\rangle)$$
$$P_2 = (\nu r)\,(\textbf{init}(x).\overline{s}\langle 100, x, r\rangle \mid r_{\mathbf{c}}().\,\textbf{close}(x)) \quad \Phi = (IR^*C\downarrow)^{\#}$$

Let $A_1 = \mu\alpha.(\overline{r} \oplus (\nu r')\,(\langle r'/r\rangle\alpha | r'_{\mathbf{c}}.\,x^R.\overline{r})$ and let $\Gamma = s{:}\textbf{chan}\langle(n{:}\textbf{int},\, x{:}\textbf{res},\, r{:}\textbf{chan}\langle\rangle)\,A_1\rangle$. Then

$$\Gamma, n{:}\textbf{int}, x{:}\textbf{res}, r{:}\textbf{chan}\langle\rangle \rhd P_1 : A_1 \quad \Gamma \rhd *s(n,x,r).\,P_1 : *s.\,(A_1{\uparrow_{\{n,x,r\}}}) \approx *s$$
$$\Gamma \rhd P_2 : (\nu r)\,(x^I.A_1 | r_{\mathbf{c}}.\,x^C)$$

So long as $\textbf{etraces}_x((\nu r)\,(x^I.A_1|r.\,x^C)) \subseteq \Phi$, we obtain $\emptyset \rhd P : \mathbf{0}$. See Section 4.1 for the algorithm that establishes $\textbf{etraces}_x(\cdot) \subseteq \Phi$. □

Remark 2. The type A_1 in the example above demonstrates how recursion, hiding, and renaming are used together. In general, in order to type a recursive process of the form $*s(x).\,(\nu y)\,(\cdots \overline{s}\langle y\rangle \cdots)$, we need to find a type that satisfies $(\nu y)\,(\cdots \langle y/x\rangle A \cdots) \;\leq\; A$. Moreover, for the type inference (in Section 4), we must find the *least* such A. Thanks to the type constructors for recursion, hiding, and renaming, we can always do that: A can be expressed by $\mu\alpha.(\nu y)\,(\cdots \langle y/x\rangle\alpha \cdots)$.

The following theorem states that no well-typed process performs an invalid access to a resource.

Theorem 1 (type soundness (safety)). *Suppose that P is safe. If $\Gamma \rhd P : A$ and $P \longrightarrow^* Q$, then Q is safe.*

Theorem 2 below states that well-typed programs eventually perform all the necessary resource accesses (unless the whole process diverges).

Definition 5 (well-annotatedness). *P is* active *if* $P \preceq (\widetilde{\nu \mathfrak{N}})(\overline{x}_{\mathbf{c}}\langle \widetilde{v}\rangle .\, Q \,|\, R)$ *or* $P \preceq (\widetilde{\nu \mathfrak{N}})(x_{\mathbf{c}}(\widetilde{y}).\, Q \,|\, R)$. *$P$ is* well-annotated *if for any P' such that $P \longrightarrow^* P'$ and active(P'), there exists P'' such that $P' \longrightarrow P''$.*

Theorem 2. *If well_annotated(P) and $\emptyset \triangleright P : A$, then P is partially live.*

4 Type Inference Algorithm

This section discusses an algorithm which takes a closed process P as an input and checks whether $\emptyset \triangleright P : \mathbf{0}$ holds. The algorithm consists of the following steps.

1. Extract constraints on type variables based on the (syntax-directed version of) typing rules.
2. Reduce constraints to trace inclusion constraints of the form $\{\mathbf{etraces}_{x_1}(A_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_n}(A_n) \subseteq \Phi_n\}$
3. Decide whether the trace inclusion constraints are satisfied.

The algorithm for Step 3 is sound but not complete.

The first two steps are fairly standard [9, 10]. Based on the typing rules, we can transform $\emptyset \triangleright P : \mathbf{0}$ to equivalent constraints of the form:

$$\{\alpha_1 \geq A_1, \ldots, \alpha_n \geq A_n, \mathbf{etraces}_{x_1}(B_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_m}(B_m) \subseteq \Phi_m\}$$

where $\alpha_1, \ldots, \alpha_n$ are different from each other. Each subtype constraint $\alpha \geq A$ can be replaced by $\alpha \geq \mu\alpha.A$. Therefore, the above constraints can be further reduced to:

$$\{\mathbf{etraces}_{x_1}([\widetilde{A'}/\widetilde{\alpha}]B_1) \subseteq \Phi_1, \ldots, \mathbf{etraces}_{x_m}([\widetilde{A'}/\widetilde{\alpha}]B_m) \subseteq \Phi_m\}$$

Here, $A'_1, \ldots, A'_n$ are the least solutions for the subtype constraints. Thus, we have reduced type checking to the validity of trace inclusion constraints of the form $\mathbf{etraces}_x(A) \subseteq \Phi$.

Example 3. Recall Example 2. We obtain the constraint $\mathbf{etraces}_x(A_1) \subseteq (IR^*C)^\#$ where

$$A_1 = (\nu r)\,(x^I.\overline{s}.\, A_2 \,|\, r.\, x^C) \qquad A_3 = \mu\alpha_2.\alpha_2$$
$$A_2 = \mu\alpha_1.\overline{r}.\, A_3 \oplus (\nu r')\,(\overline{s}.\, \langle r'/r\rangle\alpha_1 \,|\, r'.\, x^R.\overline{r}.\, A_3)\downarrow_{\{n,x,r\}}.$$

4.1 Step 3: Constraint Solving

We present an approximate algorithm for checking how to check a trace inclusion constraint $\mathbf{etraces}_x(A) \subseteq \Phi$ when the trace set Φ is a regular language. (Actually, we can extend the algorithm to deal with the case where Φ is a deterministic Petri net language: see the full version [15].)

The algorithm consists of the following three steps.

- Approximate the behavior of $A{\downarrow_{\{x\}}}$ by a (labeled) Petri net $N_{A_1,x}$.
- Construct a Petri net $N_{A'_1,x} \parallel M_\Phi$ that simultaneously simulates $N_{A_1,x}$ and a minimized deterministic automaton M_Φ that accepts Φ.
- Check that $N_{A'_1,x} \parallel M_\Phi$ does not reach any invalid state. Here, the set of invalid states consists of (1) states where $N_{A_1,x}$ can make a ξ-transition while M_Φ cannot, and (2) states where $N_{A_1,x}$ is disabled (in other words, can make a $\downarrow$-transition) while M_Φ cannot make a $\downarrow$-transition.

The last part amounts to solving a reachability problem of Petri nets. In the implementation, we further approximate the Petri net by a finite state machine.

We sketch the first step of the algorithm with an example below. Attributes are omitted below for simplicity. Please consult the full version [15] for more details and the other two steps. In Example 3 above, we have reduced the typability of the process to the equivalent constraint $\mathbf{etraces}_x(A_1) \subseteq \Phi$ where $\Phi = (IR^*C\downarrow)^\#$ and

$$A_1{\downarrow_{\{x\}}} \approx (\nu r)\,(x^I.A''_2 \mid r.\,x^C) \qquad A''_2 = \overline{r} \oplus (\nu r')\,(\langle r'/r\rangle A''_2 \mid r'.\,x^R.\overline{r})$$

Here, we have omitted $A_3 = \mu\alpha.\alpha$ since it is insignificant.

Approximate the behavior of $A_1{\downarrow_{\{x\}}}$ by a Petri net [19] $N_{A_1,x}$. This part is similar to the translation of usage expressions into Petri nets in Kobayashi's previous work [10, 11, 14]. Since the behavioral types are more expressive (having recursion, hiding, and renaming), however, we need to approximate the behavior of a behavioral type unlike in the previous work. In this case $A_1{\downarrow_{\{x\}}}$ is infinite. To make it tractable we make a sound approximation A'_1 by pushing (ν) to top level, and we eliminate $\langle r'/r\rangle$:

$$A'_1 = (\nu r, r')\,(x^I.A'_2 \mid r.\,x^C) \quad A'_2 = \overline{r} \oplus (A'_3 \mid r'.\,x^R.\overline{r}) \quad A'_3 = \overline{r'} \oplus (A'_3 \mid r'.\,x^R.\overline{r'})$$

Then $N_{A'_1,x}$ is as pictured in Figure 4. (Here we treat $A_1 \oplus A_2$ as $\tau.A_1 \oplus \tau.A_2$ for clarity. We also use a version of Petri nets with labeled transitions.) The rectangles are the places of the net, and the dots labeled by τ, x^R, etc. are the

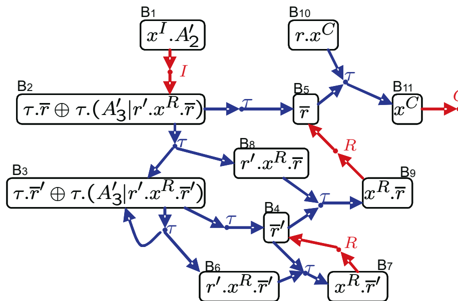


Fig. 4. $N_{A'_1,x}$

transitions of the net. Write i_x for the number of tokens at node B_x. The behavior A_1' corresponds to the initial marking $\{i_1{=}1,\ i_{10}{=}1\}$. We say that the nodes $\widetilde{B}$ together with the restricted names (r, r') constitute a *basis* for A_1'. Note here that $\mathbf{etraces}_x(A_1) \subseteq \mathbf{etraces}_x(A_1') = \mathbf{ptraces}(N_{A_1',x})$ where $\mathbf{ptraces}(N_{A_1',x})$ is the set of traces of the Petri net. Thus, $\mathbf{ptraces}(N_{A_1',x}) \subseteq \Phi$ is a sufficient condition for $\mathbf{etraces}_x(A_1) \subseteq \Phi$. The key point here is that A_1' still has infinite states, but all its reachable states can be expressed in the form $(\nu r, r')\,(i_1 B_1 \mid \cdots \mid i_{11} B_{11})$ (where $i_k B_k$ is the parallel composition of i_k copies of B_k), a linear combination of finitely many processes $\widetilde{B}$. That is why we could express A_1' by the Petri net as above.

5 Implementation

We have implemented a prototype resource usage analyzer based on the type system proposed in this paper. We have tested all the examples given in the present paper. The implementation can be tested at `http://www.yl.is.s.u-tokyo.ac.jp/~kohei/usage-pi/`.

The analyzer takes a pi-calculus program as an input, and uses `TyPiCal`[11] to annotate each input or output action with an attribute on whether the action is guaranteed to succeed automatically. The annotated program is then analyzed based on the algorithm described in Section 4.

The followings are some design decisions we made in the current implementation. We restrict the resource usage specification (Φ) to the regular languages although in future we may extend it to deterministic Petri net languages. In the algorithm for checking $\mathbf{etraces}_x(A) \subseteq \Phi$, we blindly approximate A by pushing all of its ν-prefixes to the top-level. In future we might utilize an existing model checker to handle the case where A is already finite. To solve the reachability problems of Petri nets, we approximate the number of tokens in each place by an

Input:

```
new create,s in
  *(create?(r).newR {init(read|write)*close }, x in acc(x,init).r!(x))
| *(new r in create!(r) | r?(y).new c in s!(false,y,c)
             | s!(false,y,c) | c?().c?().acc(y,close))
| *(s?(b,x,r).if b then r!() else acc(x,read).s!(b,x,r))
```

Output:

```
(*** The result of lock-freedom analysis ***)
new create, s in
  *create??(r). newR {init(read|write)*close}, x in acc(x, init). r!!(x)
| *(new r in create!!(r) | r??(y).new c in s!!(false,y,c)
             | s!!(false,y,c) | c??().c??().acc(y,close))
...
No error found
```

Fig. 5. A Sample Run of the Analyzer

element of the finite set $\{0, 1, 2, \text{“3 or more”}\}$. That approximation reduces Petri nets to finite state machines, so we can use BDD to compute an approximation of the reachable states.

Figure 5 shows a part of a successful run of the analyzer. The first process (on the second line) of the input program runs a server, which returns a new, initialized resource. We write `!` and `?` for output and input actions. The resource access specification is here expressed by a regular expression `init(read|write)*close`. The second process runs infinitely many client processes, each of which sends a request for a new resource, and after receiving it, reads and closes it. The third process (on the 6th line) is a tail-recursive version of the replicated service in Example 2. Here, a boolean is passed as the first argument of s instead of an integer, as the current system is not adapted to handle integers; it does not affect the analysis, since the system ignores the value and simply inspects both branches of the conditional. Note that the program creates infinitely many resources and has infinitely many states. The first output is the annotated version of the input program produced by `TyPiCal`, where `!!` and `??` are an output and an input with the attribute **c**.

6 Related Work

Resource usage analysis and similar analyses have recently been studied extensively, and a variety of methods from type systems to model checking have been proposed [1, 5–7, 9, 16, 20]. However, only a few of them deal with concurrent languages. To our knowledge, none of them deal with the partial liveness property. Nguyen and Rathke [18] propose an effect-type system for a kind of resource usage analysis for functional languages extended with threads and monitors. In their language, neither resources nor monitors can be created dynamically. On the other hand, our target language is π-calculus, so that our type system can be applied to programs that may create infinitely many resources (due to the existence of primitives for dynamic creation of resources: recall the example in Figure 5), and also to programs that use a wide range of communication and synchronization primitives.

Model checking technologies [2, 4, 21, 22] can of course be applicable to concurrent languages, but naive applications of model checking technologies would suffer from the state explosion problem, especially for expressive concurrent languages like π-calculus, where resources and communication channels can be dynamically created and passed around. Actually, our type-based analysis can be considered as a kind of abstract model checking. The behavioral types extracted by (the first two steps of) the type inference algorithm are abstract concurrent programs, each of which captures the access behavior on each resource. Then, conformance of the abstract program with respect to the resource usage specification is checked as a model checking problem. From that perspective, a nice point about our approach is that our type, which describes a resource-wise behavior, has much smaller state space than the whole program. In particular, if infinitely many resources are dynamically created, the whole program has in-

finite states, but it is often the case that our behavioral types are still finite (indeed so for the example in Figure 5).

Technically, closest to our type system are that of Igarashi and Kobayashi [8] and that of Chaki, Rajamani, and Rehof [3]. Those type systems are developed for checking the communication behavior of a process, but by viewing a set of channels as a resource, it is possible to use those type systems directly for the resource usage analysis. As mentioned in Section 1, the main contributions of the present work with respect to those type systems are realization of automatic verification while keeping enough precision, and verification of the partial liveness. The parameterization of the type system with an arbitrary mechanism to guarantee deadlock-freedom opens a new possibility of integrating type-based techniques with other verification techniques (the current implementation uses another type-based analyzer to infer deadlock-freedom, but one can replace that part with a model checker or an abstract interpreter).

7 Conclusion

We have presented a type-based technique for verifying resource usage of concurrent programs. Future work includes more serious assessment of the effectiveness of our analysis and extensions of the type system to deal with other typical synchronization primitives like join-patterns and external choice.

Acknowledgments

We would like to thank Andrew Gordon, Jakob Rehof, and Eijiro Sumii for useful discussions and comments. We would also like to thank anonymous referees for useful comments and suggestions.

References

1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods 2004*, volume 2999 of *LNCS*, pages 1–20. Springer-Verlag, 2004.
2. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. of POPL*, pages 1–3, 2002.
3. S. Chaki, S. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Proc. of POPL*, pages 45–57, 2002.
4. M. Dam. Model checking mobile processes. *Information and Computation*, 129(1):35–51, 1996.
5. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. of PLDI*, pages 59–69, 2001.
6. R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proc. of PLDI*, 2002.
7. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. of PLDI*, pages 1–12, 2002.

8. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.
9. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Prog. Lang. Syst.*, 27(2):264–313, 2005. Preliminary summary appeared in Proceedings of POPL 2002.
10. N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*. to appear.
11. N. Kobayashi. TyPiCal: A type-based static analyzer for the pi-calculus. Tool available at `http://www.kb.ecei.tohoku.ac.jp/~koba/typical/`.
12. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Prog. Lang. Syst.*, 20(2):436–482, 1998.
13. N. Kobayashi. A type system for lock-free processes. *Info. Comput.*, 177:122–159, 2002.
14. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proc. of CONCUR2000*, volume 1877 of *LNCS*, pages 489–503. Springer-Verlag, August 2000.
15. N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the pi-calculus. Full version, 2005. `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/usage-pi.pdf`.
16. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, volume 2895 of *LNCS*, pages 212–229, 2003.
17. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
18. N. Nguyen and J. Rathke. Typed static analysis for concurrent, policy-based, resource access control. draft.
19. J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.
20. C. Skalka and S. Smith. History effects and verification. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *LNCS*, pages 107–128, 2004.
21. B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *CAV'94: Computer Aided Verification*, volume 818 of *LNCS*, pages 428–440. Springer-Verlag, 1994.
22. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. In *Proceedings of VMCAI 2003*, volume 2575 of *LNCS*, pages 116–131. Springer-Verlag, 2003.

Semantic Hierarchy Refactoring by Abstract Interpretation

Francesco Logozzo[1] and Agostino Cortesi[2]

[1] École Normale Supérieure, France
[2] Università Ca' Foscari di Venezia, Italy

Abstract. A semantics-based framework is presented for the definition and manipulation of class hierarchies for object-oriented languages. The framework is based on the notion of observable of a class, i.e., an abstraction of its semantics when focusing on a behavioral property of interest. We define a semantic subclass relation, capturing the fact that a subclass preserves the behavior of its superclass up to a given (tunable) observed property. We study the relation between syntactic subclass, as present in mainstream object-oriented languages, and the notion of semantic subclass. The approach is then extended to class hierarchies, leading to a semantics-based modular treatment of a suite of basic observable-preserving operators on hierarchies. We instantiate the framework by presenting effective algorithms that compute a semantic superclass for two given classes, that extend a hierarchy with a new class, and that merge two hierarchies by preserving semantic subclass relations.

1 Introduction

In the object-oriented paradigm, a crucial role is played by the notion of class hierarchy. Being A a subclass of B captures the fact that the state and the behavior of the elements of A are coherent with the intended meaning of B, while disregarding the additional features and functionalities that characterize the subclass.

The approach of mainstream object-oriented languages, like Java and C++, to class hierarchies can be seen as merely syntactic. In such a view hierarchies are collections of classes ordered by the transitive closure of explicitly declared subclass or subtype relations. This is why the main theoretical and practical contributions to hierarchy refactoring issues [32, 33] combine static and dynamic analyses that focus only on syntactic elements. However, as pointed out by [29], this approach has severe limitations, as it leads to troubles when trying to face the issue of extending a given class hierarchy.

In this paper we adopt an alternative, semantics-based approach for the definition and manipulation of class hierarchies. It uses previous works on abstract interpretation theory [10], that allows formalizing the notion of different levels of property abstraction and of abstract semantics. This framework is based on the notion of observable of a class, i.e., an abstraction of the class semantics that focuses on a behavioral property of interest. The intuition is that the semantics of a class can be abstracted by parameterizing it with respect to a given domain

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 313–331, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of observables, and that a notion of semantic subclass can then be defined in terms of preservation of these observables. This notion of semantic subclass can be seen as a proper generalization of the concept of class subtyping, having the advantage of being tunable with respect to a given underlying abstract domain and hence of the properties we are interested to capture.

The notion of syntactic subclass, forcing that fields and methods have the same names, is too weak to state something about semantic subclassing, but compatibility results on the syntactic extension on one hand, and suitable renaming functions on the other can be stated that allow us to properly relate the two subclass relations.

The interest of the notion of semantic subclass become even more interesting when facing the problem of manipulating class hierarchies which has more than thousands of classes (for instance, NetBeans [27] is made up of 8328 classes). We formalize the notion of semantic ordering of hierarchies as "*when is it the case that a hierarchy is more informative with respect to a given observable?*"

We show that this notion of semantic subclassing

- can be formally related to the traditional syntactic-based subclassing relation;
- it is crucial for designing automatic and modular verification tools for polymorphic code;
- it enlightens the trade-off between the expressive power of specification languages for object-oriented languages and the subtype relations they support;
- it is the base to design algorithms and tools for extending, refactoring and merging class hierarchies.

In fact, in the paper we show how it can be used for the automatic and modular verification of polymorphic code, for bounding the expressive power of specification languages for object-oriented languages and for the characterization of semantic class hierarchies. Intuitively, semantic class hierarchies ensure that, up to a given observable, classes lower in the hierarchy specializes the behavior of the upper classes. We instantiate our framework by design algorithms for extending, refactoring and merging class hierarchies. Such algorithms represent the basis for our mid-term goal, that is a tool for the modular verification and the semi-automatic refactoring of large class hierarchies.

Paper Structure. In Section 2, an example introduces the main ideas of the paper. In Section 3, the notion of observable is introduced as an abstraction of the concrete semantics. In Section 4, we introduce the semantic subclass relation, we discuss its relationship with the syntactic notion, and we show its use for modular verification of polymorphic code. In Section 5, the framework is lifted to class hierarchies by introducing a suite of refactoring operators. Finally, Section 6 discusses related work, and Section 7 concludes.

2 A Motivating Example

Let us consider the five classes described in Fig. 1 that encode different sets of integer numbers. In class `Even`, variable `x` can only take even values, whereas variable `x`

```
class Integer {        class Even {          class  Odd {
 int x;                 int x;                int x;
 init(){ x  = 0 }       init(){ x  = 0 }      init(){ x  = 1 }
 add() { x += 1 }       add() { x += 2 }      add() { x += 2 }
 sub() { x -= 1 } }     sub() { x -= 2 } }    sub() { x -= 2 } }

class  MultEight{      class  MultTwelve{
 int x;                 int x;
 init(){ x  = 0 }       init(){ x  = 0 }
 add() { x += 16 }      add() { x += 24 }
 sub() { x -= 8 } }     sub() { x -= 12 } }
```

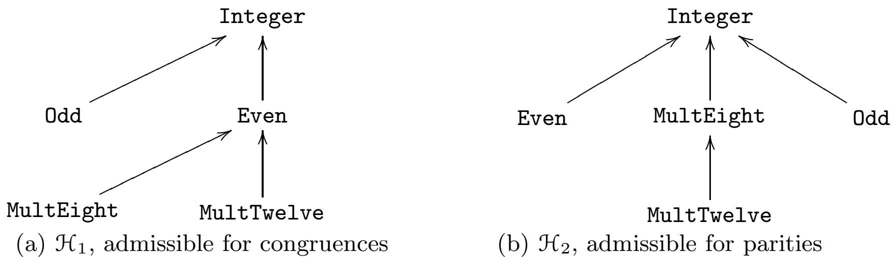
Fig. 1. Running examples

of `Odd` takes odd values only. The instance variable of `MultEight` and `MultTwelve` can only be assigned a value that is a multiple of 8 and 12, respectively.

A first question to address is "*What are the admissible hierarchies among such classes?*". A hierarchy is admissible when the subclasses preserve a given property of their superclass. So, when the *parity* of the field `x` is observed, both the class hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$ in Fig. 2 are admissible. This is true for $\mathcal{H}_1$, as the value of `MultEight.x` is always a multiple of 8, and in particular it is even. As a consequence, when just parity is observed, `MultEight` preserves the behavior of `Even`. On the other hand, $\mathcal{H}_2$ is also an admissible class hierarchy w.r.t. parity as the values taken by `MultTwelve.x` and `MultEight.x` are even numbers, too. As a consequence, `MultTwelve` preserves the parity behavior of its superclass `MultEight`. Nevertheless, when we consider a more precise property, for instance the value taken by `x` up to a numerical *congruence*, then $\mathcal{H}_2$ is no longer an admissible hierarchy. In fact, as in general a multiple of 12 is not a multiple of 8, `MultTwelve` does not preserve the congruence property of its ancestor `MultEight`.

"*Why do we need admissible class hierarchies?*" For two reasons: (i) it allows one to design modular verification tools of polymorphic methods, and (ii) it supports design of semantics-preserving operations on class hierarchies.

To illustrate (i), consider the class hierarchy $\mathcal{H}_1$ and the method `inv`, defined as follows:



(a) $\mathcal{H}_1$, admissible for congruences (b) $\mathcal{H}_2$, admissible for parities

Fig. 2. $\mathcal{H}_1$ and $\mathcal{H}_2$, two possible class hierarchies

$$\texttt{inv(Even e)}\{\texttt{return } 1/(1 - \texttt{e.x}\%2)\}.$$

In order to prove that `inv` never performs a division by zero, it suffices to prove it w.r.t. `Even` instances. In fact as $\mathcal{H}_1$ is admissible for parity, then all the subclasses of `Even` preserve the property that `x` is an even number. Nevertheless, in order to prove it correct also for all the future extensions of the hierarchy, we need to assure that all the manipulations on class hierarchies preserve its admissibility. This leads to (ii).

This semantic approach can be used to define, and prove correct, manipulating operations on class hierarchies that preserve admissibility w.r.t. a given property. For instance, we will show an algorithm for class insertion. Such an algorithm, when applied to the classes of Fig. 3 and to the hierarchy $\mathcal{H}_1$, returns the hierarchy $\mathcal{H}_3$ in Fig. 4, which is still admissible for congruences (and hence parities). As a consequence, the method `inv` is still guaranteed to be correct for all possible inputs.

```
class  MultFour {      class  MultTwenty {
 int x;                 int x;
 init() { x  = 0 }      init() { x  = 0 }
 add()  { x += 4 }      add()  { x += 20 }
 sub()  { x -= 4 }}     sub()  { x -= 60 }}
```

Fig. 3. Two classes to be added to $\mathcal{H}_1$

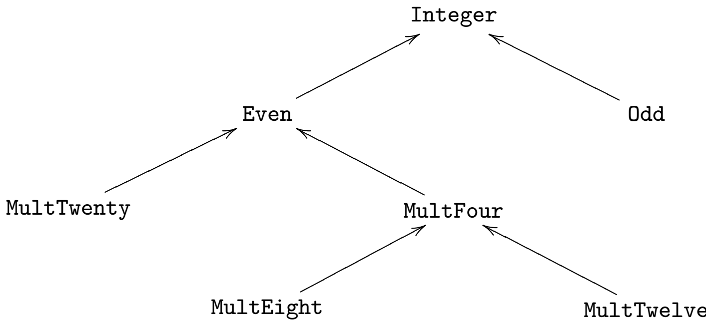


Fig. 4. $\mathcal{H}_3$: the hierarchy $\mathcal{H}_1$ augmented with `MultTwenty` and `MultFour`

3 Concrete and Abstract Semantics of Classes

In this section, we introduce the syntax and the concrete semantics of classes. Then, we define the domain of observables and the abstract semantics of a class.

3.1 Syntax

A class is a template for objects. It is provided by the programmer who specifies the fields, the methods and the class constructor.

Definition 1 (Classes). *A class* $\mathtt{C}$ *is a triple* $\langle \mathtt{F}, \mathtt{init}, \mathtt{M} \rangle$ *where* $\mathtt{F}$ *is a set of distinct variables,* $\mathtt{init}$ *is the class constructor and* $\mathtt{M}$ *is a set of method definitions. The set of all the classes is denoted by* $\mathcal{C}$.

Like in Smalltalk [15], methods are untyped and fields are private. This is just to simplify the exposition and it does not cause any loss of generality: any external access to a field `f` can be simulated by a pair of methods `set_f`/ `get_f`. Furthermore, we assume that a class has only one constructor. The generalization to an arbitrary number of constructors is straightforward. The interface of a class is the set of messages it can answer:

Definition 2 (Class Interface). *Given a class* $\mathtt{C} = \langle \mathtt{init}, \mathtt{M} \rangle$*, let* $\mathtt{M}_{\mathtt{names}}$ *be the names of* $\mathtt{C}$*'s methods. Then the interface of* $\mathtt{C}$ *is* $\iota(\mathtt{C}) = \{\mathtt{init}\} \cup \mathtt{M}_{\mathtt{names}}$.

3.2 Concrete Semantics

Given a class $\mathtt{C} = \langle \mathtt{F}, \mathtt{init}, \mathtt{M} \rangle$, every instance of $\mathtt{C}$ has an internal state $\sigma \in \Sigma$ that is a function from fields to values, i.e., $\Sigma = [\mathtt{F} \to D_{val}]$, where D_{val} is the semantic domain of values.

When a class is instantiated, the class constructor is called to set the internal state of the new object. This is modeled by a semantic function $\mathbb{i}[\![\mathtt{init}]\!] \in [D_{val} \to \mathcal{P}(\Sigma)]$. We consider sets in order to model non-determinism, e.g., user input or random choices.

The semantics of a method $\mathtt{m}$ is a function $\mathbb{m}[\![\mathtt{m}]\!] \in [D_{val} \times \Sigma \to \mathcal{P}(D_{val} \times \Sigma)]$. A method is called with two parameters: the method actual parameters and the internal state of the object it belongs to. The output of a method is a set of pairs $\langle$ return value (if any), new object state $\rangle$.

The most precise state-based property of a class $\mathtt{C}$ is the set of states reached by any execution of every instance of $\mathtt{C}$ in any possible context. In this paper, we consider just state-based properties. Such an approach can be shown to be an abstraction of a trace-based semantics for object-oriented languages, [23, 22], in which just the states before and after the invocation of a method are retained.

The set of states reached by any execution of any instance of a class can be expressed as a least fixpoint on the complete boolean lattice $\langle \mathcal{P}(\Sigma), \subseteq \rangle$. The set of the initial states, i.e., the states reached after any invocation of the $\mathtt{C}$ constructor, is:

$$S_0 = \{\sigma \in \Sigma \mid \exists v \in D_{val}.\ \sigma \in \mathbb{i}[\![\mathtt{init}]\!](v)\}.$$

The states reached after the invocation of a method $\mathtt{m}$ are given by the method collecting *forward* semantics $\mathbb{M}^{>}[\![\mathtt{m}]\!] \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$:

$$\mathbb{M}^{>}[\![\mathtt{m}]\!](S) = \{\sigma' \in \Sigma \mid \exists \sigma \in S.\ \exists v \in D_{val}.\ \exists v' \in D_{val}.\ \langle v', \sigma' \rangle \in \mathbb{m}[\![\mathtt{m}]\!]\langle v, \sigma \rangle\}.$$

The class reachable states are the least solution of the following recursive equations:

$$\begin{aligned} S &= S_0 \cup \bigcup_{\mathtt{m} \in \mathtt{M}} S_{\mathtt{m}} \\ S_{\mathtt{m}} &= \mathbb{M}^{>}[\![\mathtt{m}]\!](S) \qquad \mathtt{m} \in \mathtt{M}. \end{aligned} \tag{1}$$

The above equations characterize the set of states that are reachable before and after the invocation of any method in any instance of the class. Stated otherwise, they consider all the states reached after any possible invocation, in any order, with any input values of the methods of a class. A more general situation, in which the context may update the fields of an object, e.g. , because of aliasing, is considered in [22].

The least solution of (1) w.r.t. set inclusion corresponds to a tuple $\langle \mathsf{S}, \mathsf{S}_0, \{\mathtt{m} : \mathsf{S}_{\mathtt{m}}\}\rangle$ such that S is a class invariant [21, 23, 22], and for each method $\mathtt{m}$, $\mathsf{S}_{\mathtt{m}}$ is the strongest postcondition of the method. The method preconditions can be obtained by going backward from the postconditions: given a method $\mathtt{m}$ and its postcondition, we consider the set of states from which it is possible to reach a state in $\mathsf{S}_{\mathtt{m}}$ by an invocation of $\mathtt{m}$. Formally, the collecting *backward* method semantics $\mathbb{M}^{<}[\![\mathtt{m}]\!] \in [\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$ is defined as

$$\mathbb{M}^{<}[\![\mathtt{m}]\!](S) = \{\sigma \in \Sigma \mid \exists \sigma' \in S.\ \exists v \in D_{val}.\ \exists v' \in D_{val}.\ \langle v', \sigma' \rangle \in \mathbb{m}[\![\mathtt{m}]\!]\langle v, \sigma \rangle\}.$$

and the methods preconditions are $\mathsf{B}_{\mathtt{m}} = \mathbb{M}^{<}[\![\mathtt{m}]\!](\mathsf{S}_{\mathtt{m}})$.

The concrete class semantics, i.e., the most precise property of a class [10], is the triple $\mathbb{C}[\![\mathtt{C}]\!] = \langle \mathsf{S}, \mathsf{S}_0, \{\mathtt{m} : \mathsf{B}_{\mathtt{m}} \to \mathsf{S}_{\mathtt{m}}\}\rangle$.

The use of the concrete semantics $\mathbb{C}[\![\mathtt{C}]\!]$ for the definition of the observables of a class has two drawbacks. First, in general the computation of the least fixpoint of (1) may be unfeasible and the sets S and $\mathsf{S}_{\mathtt{m}}$ and $\mathsf{B}_{\mathtt{m}}$ may not be computer-representable. Therefore, this approach is not suitable for an effective definition of semantic subclassing. Second, it is too precise, as it may differentiate classes that do not need to be distinguished. For example, let us consider two classes `StackWithList` and `StackWithArray` which implement a stack by using respectively a linked list and a resizable array. Both of them have `push` and `pop` methods. If they are observed using the concrete semantics, then the two classes are unrelated, as the internal representation of the stack is different. On the other hand, when the behavior w.r.t. to the invocation of methods is observed, they act in the same way, e.g., no difference can be made between the values returned by the respective `pop` methods: both of them return the value on the top of the stack.

In order to overcome those drawbacks we consider abstract domains that encode the relevant properties and abstract semantics that are feasible, i.e. which are sound, but not necessarily complete, abstractions of the concrete semantics.

3.3 Domain of Observables

An observable of a class `C` is an approximation of its semantics that captures some aspects of interest of the behavior of `C`. We build a domain of observables starting from an abstraction of sets of object states.

Let us consider an abstract domain $\langle P, \sqsubseteq \rangle$, which is a complete lattice, related to the concrete domain by a Galois connection [10]:

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle P, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle. \tag{2}$$

For instance, if we are interested in the linear relations between the values of the fields of the instances of C, we instantiate P with the Octagons abstract domain [26]. On the other hand if we are interested in object aliasing then we are likely to choose for P an abstract domain that captures shapes, e.g.,[30, 31].

Once $\langle P, \sqsubseteq \rangle$ is fixed, the abstract domain $\langle O[P], \sqsubseteq_o^{[P]} \rangle$ of the observables of a class is built on top of it. The elements of the abstract domain belong to the set:

$$O[P] = \{\langle \bar{S}, \bar{S}_0, \{\mathtt{m} : \langle \bar{V}_\mathtt{m}, \bar{B}_\mathtt{m} \rangle \to \bar{S}_\mathtt{m}\}\rangle \mid \bar{S}, \bar{S}_0, \bar{V}_\mathtt{m}, \bar{B}_\mathtt{m}, \bar{S}_\mathtt{m} \in P\}.$$

Intuitively, an element of $O[P]$ consists of an approximation of the class invariant, the constructor postcondition, and for each method an approximation of its precondition and postcondition. A method precondition is in turn made up of two parts, one for the method input values and the other for that internal object state. When no ambiguity arises, we write $\langle O, \sqsubseteq_o \rangle$ instead of $\langle O[P], \sqsubseteq_o^{[P]} \rangle$. We tacitly assume that if a method n is not defined in a class, then its precondition and postconditions are respectively $\top$ and $\bot$.

The partial order $\sqsubseteq_o$ on O is defined point-wise. Let $\mathtt{o}_1 = \langle \bar{I}, \bar{I}_0, \{\mathtt{m_i} : \langle \bar{U}_i, \bar{R}_i \rangle \to \bar{I}_i\}\rangle$ and $\mathtt{o}_2 = \langle \bar{J}, \bar{J}_0, \{\mathtt{m_j} : \langle \bar{W}_j, \bar{Q}_j \rangle \to \bar{J}_j\}\rangle$ be two elements[1] of O. Then the order $\sqsubseteq_o$ is defined as:

$$\mathtt{o}_1 \sqsubseteq_o \mathtt{o}_2 \iff \bar{I} \sqsubseteq \bar{J} \wedge \bar{I}_0 \sqsubseteq \bar{J}_0 \wedge (\forall \mathtt{m_i}.\ \bar{W}_i \sqsubseteq \bar{U}_i \wedge \bar{Q}_i \sqsubseteq \bar{R}_i \wedge \bar{I}_i \sqsubseteq \bar{J}_i).$$

If $\mathtt{o}_1$ and $\mathtt{o}_2$ are the observables of two classes A and B then the order $\sqsubseteq_o$ ensures that A preserves the class invariant of B and that the methods of A are a "safe" replacement of those with the same name in B. Intuitively, the precondition generalizes the observations, made in the context of type theory, of [3]. It states two things. First, if the context satisfies $\bar{W}_i$ then it satisfies the inherited method precondition $\bar{U}_i$ too (i.e., $\bar{W}_i \sqsubseteq \bar{U}_i$). Thus the inherited method can be used in any context where its ancestor can. Second, the state of $\mathtt{o}_1$ before the invocation of a method must be *compatible* with that of $\mathtt{o}_2$ (i.e., $\bar{Q}_i \sqsubseteq \bar{R}_i$). Finally, the postcondition of the inherited method must be at least as strong as that of the ancestor (i.e., $\bar{I}_i \sqsubseteq \bar{J}_i$).

Having defined $\sqsubseteq_o$, it is routine to check that $\bot_o = \langle \bot, \bot, \{\mathtt{m_i} : \langle \top, \top \rangle \to \bot\}\rangle$ is the smallest element of O and $\top_o = \langle \top, \top, \{\mathtt{m_i} : \langle \bot, \bot \rangle \to \top\}\rangle$ is the largest one. The join, $\sqcup_o$, and the meet, $\sqcap_o$, operators on O can be defined point-wise.

Suppose that the order relation $\sqsubseteq$ on P is decidable [28]. This is the case for abstract domains used for effective static analyses. As $\sqsubseteq_o$ is defined in terms of $\sqsubseteq$ and the universal quantification ranges on a finite number of methods then $\sqsubseteq_o$ is decidable too.

Theorem 1. *Let $\langle P, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ be a complete lattice. Then $\langle O, \sqsubseteq_o, \bot_o, \top_o, \sqcup_o, \sqcap_o \rangle$ is a complete lattice. Moreover, if $\sqsubseteq$ is decidable then $\sqsubseteq_o$ is decidable too.*

From basic abstract interpretation theory [11] we know that $\mathcal{A}(\mathcal{P}(\Sigma))$, the set of all the abstractions of the concrete domain, is a complete lattice ordered w.r.t.

[1] We use the same index for methods with the same name. For instance P_i and Q_i are the preconditions for the homonym method $\mathtt{m_i}$ of $\mathtt{o}_1$ and $\mathtt{o}_2$.

the "relative" precision, $\leq$, of abstract domains. As immediate consequence, we obtain that Galois connections can be lifted to the domain of observables:

Lemma 1. *Let $\langle P, \sqsubseteq \rangle$ and $\langle P', \sqsubseteq' \rangle$ be two domains in $\mathcal{A}(\mathcal{P}(\Sigma))$ such that $\langle P, \sqsubseteq \rangle \leq \langle P', \sqsubseteq' \rangle$ with the Galois connection $\langle \alpha, \gamma \rangle$. Then,*

$$\langle O[P], \sqsubseteq_o^{[P]} \rangle \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \langle O[P'], \sqsubseteq_o^{[P']} \rangle$$

where α_o and γ_o are

$$\alpha_0(\langle \bar{S}, \bar{S}_0, \{\mathtt{m} : \langle \bar{V}_\mathtt{m}, \bar{B}_\mathtt{m} \rangle \to \bar{S}_\mathtt{m}\} \rangle = \langle \alpha(\bar{S}), \alpha(\bar{S}_0), \{\mathtt{m} : \langle \alpha(\bar{V}_\mathtt{m}), \alpha(\bar{B}_\mathtt{m}) \rangle \to \alpha(\bar{S}_\mathtt{m})\} \rangle$$
$$\gamma_0(\langle \bar{S}', \bar{S}'_0, \{\mathtt{m} : \langle \bar{V}'_\mathtt{m}, \bar{B}'_\mathtt{m} \rangle \to \bar{S}'_\mathtt{m}\} \rangle = \langle \gamma(\bar{S}'), \gamma(\bar{S}'_0), \{\mathtt{m} : \langle \gamma(\bar{V}'_\mathtt{m}), \gamma(\bar{B}'_\mathtt{m}) \rangle \to \gamma(\bar{S}'_\mathtt{m})\} \rangle.$$

3.4 Abstract Semantics

Once the abstract domain is defined, an abstraction of $\mathbb{C}[\![\mathtt{C}]\!]$ can be obtained by considering the abstract counterpart for (1). As a first step we need to consider the abstraction corresponding to the initial states, and the forward and the backward collecting semantics. We consider the *best* abstract counterparts for such concrete semantic functions.

By Galois connection properties, the best approximation for the initial states of the class is $\alpha(S_0) = \bar{S}_0$. By [11], the best approximation in P of the forward collecting method semantics of $\mathtt{m}$ of $\mathtt{C}$ is $\bar{\mathbb{M}}^{>}[\![\mathtt{m}]\!] \in [P \to P]$ defined as $\bar{\mathbb{M}}^{>}[\![\mathtt{m}]\!](\bar{S}) = \alpha \circ \mathbb{M}^{>}[\![\mathtt{m}]\!] \circ \gamma(\bar{S})$. The abstract counterpart for the equations (1) is the following equation system:

$$\begin{aligned} \bar{S} &= \bar{S}_0 \sqcup \bigsqcup_{\mathtt{m} \in \mathtt{M}} \bar{S}_\mathtt{m} \\ \bar{S}_\mathtt{m} &= \bar{\mathbb{M}}^{>}[\![\mathtt{m}]\!](\bar{S}) \qquad \mathtt{m} \in \mathtt{M}. \end{aligned} \tag{3}$$

The above equations are monotonic and, by the Tarski fixpoint theorem, there exists a least solution $\langle \bar{\mathsf{S}}, \bar{\mathsf{S}}_0, \{\mathtt{m} : \bar{\mathsf{S}}_\mathtt{m}\} \rangle$. Similarly to the concrete case, the abstract preconditions can be obtained by considering the best approximation of the backward collecting method semantics $\bar{\mathbb{M}}^{<}[\![\mathtt{m}]\!] \in [P \to P]$ defined as $\bar{\mathbb{M}}^{<}[\![\mathtt{m}]\!](\bar{S}) = \alpha \circ \mathbb{M}^{<}[\![\mathtt{m}]\!] \circ \gamma(\bar{S})$. The method abstract preconditions are obtained by projecting $\bar{\mathbb{M}}^{<}[\![\mathtt{m}]\!](\bar{\mathsf{S}}_\mathtt{m})$ respectively on the method input values and the instance fields: $\bar{\mathsf{V}}_\mathtt{m} = \pi_{in}(\bar{\mathbb{M}}^{<}[\![\mathtt{m}]\!](\bar{\mathsf{S}}_\mathtt{m}))$ and $\bar{\mathsf{B}}_\mathtt{m} = \pi_\mathtt{F}(\bar{\mathbb{M}}^{<}[\![\mathtt{m}]\!](\bar{\mathsf{S}}_\mathtt{m}))$.

To sum up, the triple $\bar{\mathbb{C}}[\![\mathtt{C}]\!] = \langle \bar{\mathsf{S}}, \bar{\mathsf{S}}_0, \{\mathtt{m} : \langle \bar{\mathsf{V}}_\mathtt{m}, \bar{\mathsf{B}}_\mathtt{m} \rangle \to \bar{\mathsf{S}}_\mathtt{m}\} \rangle$ belongs to the domain of observables, and it is the best sound approximation of the semantics of $\mathtt{C}$, w.r.t the properties encoded by the abstract domain $\langle P, \sqsubseteq \rangle$.

Theorem 2 (Observable of a Class). *Let $\langle P, \sqsubseteq \rangle$ be an abstract domain that satisfies (2) and let the observable of a class $\mathtt{C}$ w.r.t. the property encoded by $\langle P, \sqsubseteq \rangle$ be $\bar{\mathbb{C}}[\![\mathtt{C}]\!] = \langle \bar{\mathsf{S}}, \bar{\mathsf{S}}_0, \{\mathtt{m} : \langle \bar{\mathsf{V}}_\mathtt{m}, \bar{\mathsf{B}}_\mathtt{m} \rangle \to \bar{\mathsf{S}}_\mathtt{m}\} \rangle$. Then $\alpha_o(\mathbb{C}[\![\mathtt{C}]\!]) \sqsubseteq_o \bar{\mathbb{C}}[\![\mathtt{C}]\!]$.*

Example 1. Let us instantiate $\langle P, \sqsubseteq \rangle$ with Con, the abstract domain of equalities of linear congruences, [16]. The elements of such a domain have the form

$\mathtt{x} = a \bmod b$, where $\mathtt{x}$ is a program variable and a and b are integers. The representation function $\gamma_c \in [\mathsf{Con} \to \mathcal{P}(\Sigma)]$ is defined as

$$\gamma_c(\mathtt{x} = a \bmod b) = \{\sigma \in \Sigma \mid \exists k \in \mathbb{N}.\ \sigma(\mathtt{x}) = a + k \cdot b\}.$$

Let us consider the classes `Even` and `MultEight` in Fig. 2 and let e be the property $\mathtt{x} = 0 \bmod 2$, d the property $\mathtt{x} = 1 \bmod 2$ and u be the property $\mathtt{x} = 0 \bmod 8$. Then the observables of `Even` and `MultEight` w.r.t. Con are

$$\begin{aligned}
\bar{\mathbb{C}}[\![\mathtt{Even}]\!] &= \langle e, e, \{\mathsf{add} : \langle \bot, e \rangle \to e, \mathsf{sub} : \langle \bot, e \rangle \to e\} \rangle \\
\bar{\mathbb{C}}[\![\mathtt{Odd}]\!] &= \langle d, d, \{\mathsf{add} : \langle \bot, d \rangle \to d, \mathsf{sub} : \langle \bot, d \rangle \to d\} \rangle \\
\bar{\mathbb{C}}[\![\mathtt{MultEight}]\!] &= \langle u, u, \{\mathsf{add} : \langle \bot, u \rangle \to u, \mathsf{sub} : \langle \bot, u \rangle \to u\} \rangle.
\end{aligned}$$

It is worth noting that as `add` and `sub` do not have an input parameter, the corresponding precondition for the input values is $\bot$. □

4 Subclassing

The notion of subclassing can be defined both at semantic and syntactic level. Given two classes `A` and `B`, `A` is a *syntactic* subclass of `B`, denoted $\mathtt{A} \blacktriangleleft \mathtt{B}$, if all the names defined in `B` are defined in `A` too. On the other hand, `A` is a *semantic* subclass of `B`, denoted $\mathtt{A} \lhd \mathtt{B}$, if `A` preserves the observable of `B`. The notion of semantic subclassing is useful for exploring the expressive power of specification languages and the modular verification of object-oriented programs.

4.1 Syntactic Subclassing

The intuition behind the syntactic subclassing relation is inspired by the Smalltalk approach to inheritance: a subclass must answer to all the messages sent to its superclass. Stated otherwise, the syntactic subclassing relation is defined in terms of inclusion of class interfaces:

Definition 3 (Syntactic Subclassing). *Let* `A` *and* `B` *be two classes,* $\iota(\cdot)$ *as in Def. 2. Then the syntactic subclass relation is defined as* $\mathtt{A} \blacktriangleleft \mathtt{B} \iff \iota(\mathtt{A}) \supseteq \iota(\mathtt{B})$.

It is worth noting that as $\iota(\cdot)$ does not distinguish between names of fields and methods, class $\mathtt{A} = \langle \emptyset, \mathtt{init}, \mathtt{f} = \lambda \mathtt{x}.\mathtt{x} + 1 \rangle$ is a syntactic subclass of $\mathtt{B} = \langle \mathtt{f}, \mathtt{init}, \emptyset \rangle$, even if in the first case `f` is a name of a method and in the second it is the name of a field. This is not surprising in the general, untyped, context we consider.

Example 2. In mainstream object-oriented languages the subclassing mechanism is provided through class extension. For example, in Java a subclass of a base class `B` is created by using the syntactic construct "`A extends B` { extension }", where `A` is the name of the subclass and `extension` are the fields and the methods added and/or redefined by the subclass. As a consequence, if type declarations are considered part of the fields and method names, then $\mathtt{A} \blacktriangleleft \mathtt{B}$ always holds. □

4.2 Semantic Subclassing

The semantic subclassing relation formalizes the intuition that up-to a given property, a class `A` behaves like a class `B`. For example, if the property of interest is the type of the class, then `A` is a semantic subclass of `B` if its type is a subtype of `B`. In our framework, semantic subclassing can be defined in terms of the preservation of observables. In fact, as $\sqsubseteq_o$ is the abstract counterpart for the logical implication then $\bar{\mathbb{C}}[\![\mathtt{A}]\!] \sqsubseteq_o \bar{\mathbb{C}}[\![\mathtt{B}]\!]$ means that `A` preserves the semantics of `B`, when a given property of interest is observed. Therefore we can define

Definition 4 (Semantic Subclassing). *Let $\langle O, \sqsubseteq_o \rangle$ be an abstract domain of observables and let* `A` *and* `B` *be two classes. Then the semantic subclassing relation with respect to O is defined as* $\mathtt{A} \lhd_O \mathtt{B} \iff \bar{\mathbb{C}}[\![\mathtt{A}]\!] \sqsubseteq_o \bar{\mathbb{C}}[\![\mathtt{B}]\!]$.

Example 3. Let us consider the classes `Even`, `Odd` and `MultEight` and their respective observables as in Ex. 1. Then, as $u \sqsubseteq e$ holds, we have that `MultEight` $\lhd$`Even`. On the other hand, we have that neither $e \sqsubseteq d$ nor $d \sqsubseteq e$. As a consequence, `Even` $\not\lhd$ `Odd` and `Odd` $\not\lhd$ `Even`. □

Observe that when $\langle O, \sqsubseteq_o \rangle$ is instantiated with the types abstract domain [9] then the relation defined above coincides with the traditional subtyping-based definition of subclassing [2].

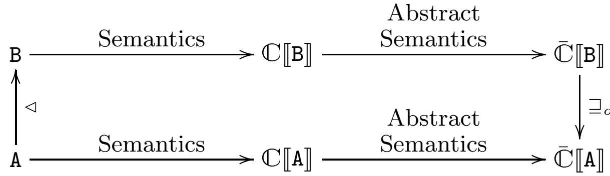


Fig. 5. A visualization of the semantic subclassing relation

The relation between classes, concrete semantics and observables can be visualized by the diagram in Fig. 5. When the abstract semantics of `A` and `B` are compared, that of `A` implies the one of `B`. This means that `A` refines `B` w.r.t. the properties encoded by the abstract domain O, in accord with the mundane approach of inheritance where a subclass is as a specialization of its ancestors [25].

The next lemma states the monotonicity of $\lhd$ w.r.t. the observed properties:

Lemma 2. *Let* `A` *and* `B` *be classes, $\langle P, \sqsubseteq \rangle$ and $\langle P', \sqsubseteq' \rangle$ be abstract domains in $\mathcal{A}(\mathcal{P}(\Sigma))$ such that $\langle P, \sqsubseteq \rangle \leq \langle P', \sqsubseteq' \rangle$. If* $\mathtt{A} \lhd_{O[P]} \mathtt{B}$ *then* $\mathtt{A} \lhd_{O[P']} \mathtt{B}$.

By Lemma 2, the more precise the domain of observables, the more precise the induced subclass relation. If we *observe* a more precise property about the class semantics then we are able to better *distinguish* between the different classes.

Example 4. Let us consider the hierarchies $\mathcal{H}_1$ and $\mathcal{H}_2$ depicted in Fig. 2. As the domain of congruences is (strictly) more precise than the domain of parities, $\mathcal{H}_1$ is also admissible for parities, by Lemma 2. Observe that in general the converse is not true: for instance $\mathcal{H}_2$ is not admissible for congruences. □

When considering the *identity* Galois connection $\langle \lambda x.\ x, \lambda x.\ x \rangle$, Def. 4 above boils down to the observation of the concrete semantics, so that by Lemma 2, $\lhd_{O[\mathcal{P}(\Sigma)]}$ is the most precise semantic subclassing relation. Furthermore, the semantic subclass relation induced by the most abstract domain is the trivial one, in which all classes are in relation with all others. As a consequence, given two classes A and B there always exist an abstract domain of observables O such that A $\lhd_O$ B. However, in general there not exists a *least* domain of observables such that the two are in the semantic subclass relation, as shown by the following example:

Example 5. Let us consider two classes A and B that are equal except for a method m defined as:

```
A.m() {
  x = 1; y = 2;
  if (x > 0) && (y % 2 == 0) {
    x = 1; y = 4; }
  else {
    x = 1; y = 8; }}
```

```
B.m() {
  x = 1; y = 2;
  if (x > 0) && (y % 2 == 0) {
    x = 1; y = 2; }
  else {
    x = 3; y = 10; }}
```

When considering the domain of intervals [10] as observables, we infer that A $\lhd_{\text{Intervals}}$ B as $([1, 1], [4, 8]) \sqsubseteq ([1, 3], [2, 10])$ and when considering the domain of parities as observables, we infer that A $\lhd_{\text{Parities}}$ B as $(\text{odd}, \text{even}) \sqsubseteq (\text{odd}, \text{even})$. In fact, in both cases the abstract domain is not precise enough to capture the branch chosen by the conditional statement. Nevertheless, when considering the reduced product, [11], Intervals $\times$ Parities we have that A $\not\lhd_{\text{Intervals}\times\text{Parities}}$ B as

$$(([1, 1], \text{odd}), ([4, 4], \text{even})) \not\sqsubseteq (([1, 1], \text{odd}), ([2, 2], \text{even})).$$

As a consequence, if there exists a least domain O such that A $\lhd_O$ B, then O should be *strictly* smaller than both Intervals and Parities as the two domains are not comparable. Then, O must be smaller or equal to the reduced product of the two domains. We have just shown that it cannot be equal. By Lemma 2 it follows that it cannot be smaller, too. □

Observation. The previous example emphasizes a strict link between the concept of subtyping in specification languages for object-oriented programs and the notion of abstraction. Let us consider two classes A and B, two specification languages $\mathcal{L}_1$ and $\mathcal{L}_2$, and the *strongest* properties we can express about the behavior of A and B in $\mathcal{L}_1$ and in $\mathcal{L}_2$, say respectively $\varphi_1^{\mathtt{A}}, \varphi_1^{\mathtt{B}}$ and $\varphi_2^{\mathtt{A}}, \varphi_2^{\mathtt{B}}$. Let us suppose that $\varphi_1^{\mathtt{A}} \Rightarrow \varphi_1^{\mathtt{B}}$ and $\varphi_2^{\mathtt{A}} \Rightarrow \varphi_2^{\mathtt{B}}$. By definition of behavioral subtyping, [18], A is a subclass of B in both $\mathcal{L}_1$ and in $\mathcal{L}_2$. Nevertheless, by Ex. 5, by the definition of observable of a class, and by basic abstract interpretation theory [8], it follows that if we consider a specification language $\mathcal{L}_3$ expressive enough to contain both $\mathcal{L}_1$ and $\mathcal{L}_2$, and the corresponding *strongest* properties $\varphi_3^{\mathtt{A}}, \varphi_3^{\mathtt{B}}$, then $\varphi_3^{\mathtt{A}} \not\Rightarrow \varphi_3^{\mathtt{B}}$. This means that when a more expressive language is used then the classes A and B are no more related. This fact enlightens an interesting trade-off between the expressive power of specification languages for object-oriented programs and the granularity of the subtyping relation they support.

4.3 Modular Verification of Polymorphic Code

Thanks to the following lemma, the notion of semantic subclass turns out to be useful for the modular verification of polymorphic code:

Lemma 3 (Modular Verification). *Let $\langle P, \sqsubseteq \rangle$ be an abstract domain, let $g \in [O[P] \to P]$ be a monotonic function and let $f \in [\mathcal{C} \to P]$ be defined as $f = \lambda \mathtt{B}.\ g(\bar{\mathbb{C}}[\![\mathtt{B}]\!])$. Then, $\mathtt{A} \lhd_{O[P]} \mathtt{B}$ implies that $f(\mathtt{A}) \sqsubseteq f(\mathtt{B})$.*

Let us consider a function "m(B b) { $\mathtt{body_m}$ }". The best abstract semantics,[11], of m w.r.t. a given abstract domain $\langle P, \sqsubseteq \rangle$ is $\bar{\mathbb{s}}[\![\mathtt{m}]\!]$ [2]. By Galois connection properties, $\bar{\mathbb{s}}[\![\mathtt{m}]\!]$ is a monotonic function. Let $o \in O[P]$. We define g as the function obtained from $\bar{\mathbb{s}}[\![\mathtt{m}]\!]$ by replacing each occurrence of an invocation of a method of b, e.g. b.n(...), inside $\mathtt{body_m}$ with the corresponding preconditions and postconditions of o [14]. We denote it with $\mathtt{m}[\mathtt{b} \mapsto \mathtt{o}]$. Hence, $g = \lambda o.\ \bar{\mathbb{s}}[\![\mathtt{m}[\mathtt{b} \mapsto \mathtt{o}]]\!]$ is a monotonic function, and in particular $\bar{\mathbb{s}}[\![\mathtt{m}]\!] \sqsubseteq g(\bar{\mathbb{C}}[\![\mathtt{B}]\!])$ as $\bar{\mathbb{C}}[\![\mathtt{B}]\!]$ is an approximation of the behavior of b in all the possible contexts. Then, we can apply Lemma 3 so that for every class A, $\mathtt{A} \lhd_{O[P]} \mathtt{B}$, we have that $g(\bar{\mathbb{C}}[\![\mathtt{A}]\!]) \sqsubseteq g(\bar{\mathbb{C}}[\![\mathtt{B}]\!])$. As a consequence, if we can prove that $g(\bar{\mathbb{C}}[\![\mathtt{B}]\!]) \sqsubseteq \bar{\mathsf{S}}$ for a given specification $\bar{\mathsf{S}}$, by transitivity, it follows that $g(\bar{\mathbb{C}}[\![\mathtt{A}]\!]) \sqsubseteq \bar{\mathsf{S}}$, for every semantic subclass $\mathtt{A} \lhd_{O[P]} \mathtt{B}$, i.e., m is correct w.r.t. the specification $\bar{\mathsf{S}}$.

Example 6. Consider the function inv in Sect. 2. We want to prove the property that inv never performs a division by zero. Let us instantiate P with the parity abstract domain. By Ex. 1 we know that $\mathtt{x} = e$. By an abstract evaluation of the return expression, one obtains $1/(1 - e\%2) = 1/d$, that is always defined (as obvisiously zero is not an odd number). As a consequence, when an instance of Even is passed to inv, it does not throw any division-by-zero exception. Furthermore, for what said above, this is true for all the semantic subclasses of Even. □

4.4 Relation Between ◀ and ◁

Consider two classes A and B such that A ◀ B. By definition, this means that all the names (fields or methods) defined in B are defined in A too. In general, such a condition is too weak to state something "*interesting*" about the semantics of A w.r.t. that of B: as seen before, there exists a domain of observables O such that $\mathtt{A} \lhd_O \mathtt{B}$, and in most cases such a domain is the most abstract one, and by Lemma 2 this implies that ◁ is a uninteresting relation. Therefore, in order to obtain more interesting subclass relations, we have to consider some hypotheses on the abstract semantics of the methods of the class. If the constructor of a class A is *compatible* with that of B, and if the methods of A do not violate the class invariant of B, then A is a semantic subclass of B. On the other hand, semantic subclassing *almost* implies syntactic subclassing. This is formalized by the following theorems [24]:

[2] We consider the best abstract function in order to simplify the exposition. Nevertheless the paper's results still hold when a generic upper-approximation is considered.

Theorem 3. *Let* $\mathtt{A} = \langle \mathtt{F_A}, \mathtt{init_A}, \mathtt{M_A} \rangle$ *and* $\mathtt{B} = \langle \mathtt{F_B}, \mathtt{init_B}, \mathtt{M_B} \rangle$ *be two classes such that* $\mathtt{A} \blacktriangleleft \mathtt{B}$*, and let* $\langle P, \sqsubseteq \rangle \in \mathcal{A}(\mathcal{P}(\Sigma))$*. If (i)* $I_\mathtt{B}$ *is a class invariant for* $\mathtt{B}$*, (ii)* $\bar{\mathbb{M}}^{>}[\![\mathtt{init_A}]\!] \sqsubseteq \bar{\mathbb{M}}^{>}[\![\mathtt{init_B}]\!]$*, (iii)* $\forall \bar{S} \in P.\ \forall \mathtt{m} \in \mathtt{M_A} \cap \mathtt{M_B}.\ \bar{\mathbb{M}}^{>}[\![\mathtt{m}]\!](\bar{S}) \sqsubseteq I_\mathtt{B}$ *and (iv)* $\forall \mathtt{m} \in \mathtt{M_A}.\ \mathtt{m} \notin \mathtt{M_B} \Longrightarrow \bar{\mathbb{M}}^{>}[\![\mathtt{m}]\!](\bar{S}) \sqsubseteq I_\mathtt{B}$ *then* $\mathtt{A} \lhd_{O[P]} \mathtt{B}$*.*

Theorem 4. *Let* $\mathtt{A}, \mathtt{B} \in \mathcal{C}$*, such that* $\mathtt{A} \lhd_O \mathtt{B}$*. Then there exists a renaming function* ϕ *such that* $\phi(\mathtt{A}) \blacktriangleleft \mathtt{B}$*.*

5 Meaning-Preserving Manipulation of Class Hierarchies

In this Section, we exploit the results of the previous sections to introduce the concept of admissible class hierarchy, and to define and prove correct some operators on class hierarchies.

5.1 Admissible Semantic Class Hierarchies

For basic definitions on trees, the reader may refer to [6]. If T is a tree, $\mathrm{nodesOf}(T)$ denotes the elements of the tree, $\mathrm{rootOf}(T)$ denotes the root of the tree, and if $n \in \mathrm{nodesOf}(T)$ then $\mathrm{sonsOf}(n)$ are the successors of the node n. In particular, if $\mathrm{sonsOf}(n) = \emptyset$ then n is a leaf. A tree with a root r and successors S is $\mathrm{tree}(r, S)$.

Here we only consider single inheritance so that class hierarchies are trees of classes. An admissible hierarchy w.r.t. a transitive relation ρ on classes is a tree such that all the nodes are classes, and given two nodes n and n' such that $n' \in \mathrm{sonsOf}(n)$ then n' is in the relation ρ with n. Formally:

Definition 5 (Admissible Class Hierarchy). *Let* $\mathcal{H}$ *be a tree and* $\rho \subseteq \mathcal{C} \times \mathcal{C}$ *be a transitive relation on classes. Then we say that* $\mathcal{H}$ *is a class hierarchy which is admissible w.r.t.* ρ*, if (i)* $\mathrm{nodesOf}(\mathcal{H}) \subseteq \mathcal{C}$*, and (ii)* $\forall n \in \mathrm{nodesOf}(\mathcal{H}).\ \forall n' \in \mathrm{sonsOf}(n). n' \rho n$*.*

We denote the set of all the class hierarchies admissible w.r.t. ρ as $\mathbb{H}[\rho]$. It is worth noting that our definition subsumes the definition of class hierarchies of mainstream object-oriented languages. In fact, when ρ is instantiated with $\blacktriangleleft$, we obtain class hierarchies in which all the subclasses have at least the same methods as their superclass. A *semantic class hierarchy* is just the instantiation of the Def. 5 with the relation $\lhd$. The theorems and lemmata of the previous sections can be easily lifted to class hierarchies:

Example 7. Consider the two hierarchies in Fig. 2. $\mathcal{H}_1$ is admissible w.r.t. $\lhd_{\mathsf{Con}}$ and $\mathcal{H}_2$ is admissible w.r.t. $\lhd_{\mathrm{Parities}}$ but not w.r.t. $\lhd_{\mathsf{Con}}$. □

In order to manipulate hierarchies we wish to preserve admissibility. This is why we need the notion of a *fair* operator. A fair operator on class hierarchies transforms a set of class hierarchies admissible w.r.t. a relation ρ into a class hierarchy that is admissible w.r.t. a relation ρ'.

Definition 6 (Fair Operator). *Let* ρ *and* ρ' *be transitive relations. Then we say that a function* $\mathtt{t}$ *is a* fair operator *w.r.t.* ρ *and* ρ' *if* $\mathtt{t} \in [\mathcal{P}(\mathbb{H}[\rho]) \to \mathbb{H}[\rho']]$*.*

In the following, when not stated otherwise, we assume that $\rho = \rho' = \lhd$.

5.2 Class Insertion

The first fair operator we consider is the one for adding a class into an admissible class hierarchy. The algorithm definition of such an operator is presented in Fig. 6. It uses as sub-routine CSS, an algorithm for computing a common semantic superclass of two given classes, that is depicted in Fig. 7 and discussed in the next section.

The insertion algorithm takes as input an admissible class hierarchy $\mathcal{H}$ and a class C. Four cases are distinguished. (i) if C already belongs to $\mathcal{H}$ then the hierarchy keeps unchanged. (ii) If C is a superclass of the root of $\mathcal{H}$, then a new class hierarchy whose root is C is returned. (iii) If C is a subclass of the root of $\mathcal{H}$, then the insertion must preserve the *admissibility* of the hierarchy. If C is a superclass of some of the successors, then it is inserted between the root of $\mathcal{H}$ and such successors. Otherwise it checks whether some root class of the successors is a superclass of C. If it is the case, then the algorithm is recursively applied, otherwise C is added at this level of the hierarchy. (iv) If C and the root of $\mathcal{H}$

$\mathcal{H} \uplus \mathtt{C} \triangleq$ **let** $\mathtt{R} = \mathrm{rootOf}(\mathcal{H})$, $\mathrm{S} = \mathrm{sonsOf}(\mathtt{R})$
  let $\mathcal{H}_< = \{\mathcal{K} \in S \mid \mathrm{rootOf}(\mathcal{K}) \lhd \mathtt{C}\}$
  let $\mathcal{H}_> = \{\mathcal{K} \in S \mid \mathrm{rootOf}(\mathcal{K}) \rhd \mathtt{C}\}$
  if $\mathtt{C} \in \mathrm{nodesOf}(\mathcal{H})$ **then return** $\mathcal{H}$
  if $\mathtt{R} \lhd \mathtt{C}$ **then return** $\mathrm{tree}(\mathtt{C}, \mathtt{R})$
  if $\mathtt{C} \lhd \mathtt{R}$ **then**
    if $\mathcal{H}_< \neq \emptyset$ **then**
      return $\mathrm{tree}(\mathtt{R}, (S\text{-}\mathcal{H}_<) \cup \mathrm{tree}(\mathtt{C}, \mathcal{H}_<))$
    if $\mathcal{H}_> \neq \emptyset$ **then select** $\mathcal{K} \in S$
      return $\mathrm{tree}(\mathtt{R}, (S\text{-}\mathcal{K}) \cup (\mathcal{K} \uplus \mathtt{C}))$
    else return $\mathrm{tree}(\mathtt{R}, S \cup \{\mathtt{C}\})$
  else select $\mathtt{C}_\top = \mathrm{CSS}(\mathtt{R}, \mathtt{C})$
    return $\mathrm{tree}(\mathtt{C}_\top, \{\mathtt{R}, \mathtt{C}\})$

Fig. 6. The algorithm for a fair class insertion

$\mathrm{CSS}(\mathtt{A}, \mathtt{B}) \triangleq$ **let** $\mathtt{A} = \langle \mathtt{F_A}, \mathtt{init_A}, \mathtt{M_A} \rangle$,
    $\mathtt{B} = \langle \mathtt{F_B}, \mathtt{init_B}, \mathtt{M_B} \rangle$,
    $\mathtt{F_C} = \emptyset$, $\mathtt{init_C} = \mathtt{init_A}$, $\mathtt{M_C} = \emptyset$
  repeat
    select $\mathtt{f} \in \mathtt{F_A} - \mathtt{F_C}$
      if $\mathtt{B} \lhd \langle \mathtt{F_C} \cup \{\mathtt{f}\}, \tau_{\mathtt{F_C} \cup \{\mathtt{f}\}}(\mathtt{init_A}), \tau_{\mathtt{F_C} \cup \{\mathtt{f}\}}(\mathtt{M_C}) \rangle$
        then $\mathtt{F_C} = \mathtt{F_C} \cup \{\mathtt{f}\}$,
          $\mathtt{init_C} = \tau_{\mathtt{F_C} \cup \{\mathtt{f}\}}(\mathtt{init_A})$
    $\|$ **select** $\mathtt{m} \in \mathtt{M_A} - \mathtt{M_C}$
      if $\mathtt{B} \lhd \langle \mathtt{F_C}, \mathtt{init_C}, \tau_{\mathtt{F_C}}(\mathtt{M_C} \cup \{\mathtt{m}\}) \rangle$
        then $\mathtt{M_C} = \mathtt{M_C} \cup \{\mathtt{m}\}$
  until no more fields or methods are added
  return $\langle \mathtt{F_C}, \mathtt{init_C}, \tau_{\mathtt{F_C}}(\mathtt{M_C}) \rangle$

Fig. 7. Algorithm for computing the CSS

are unrelated, the algorithm returns a new hierarchy whose root is a superclass of both `C` and the root of $\mathcal{H}$.

The soundness of the algorithm follows from the observation that, if in the input hierarchy there is an admissible path from a class `B` to a class `A`, then in the extended hierarchy there still exists an admissible path from `B` to `A`.

Lemma 4 (Soundness of $\uplus$, [24]). *The operator $\uplus$ defined in Fig. 6 is a fair operator w.r.t. $\lhd$, i.e., $\uplus \in [\mathbb{H}[\lhd] \times \mathcal{C} \rightarrow \mathbb{H}[\lhd]]$.*

Example 8. Consider the hierarchy $\mathcal{H}_1$ and the classes `MultFour` and `MultTwenty` of Sect.2. $(\mathcal{H}_1 \uplus \texttt{MultFour}) \uplus \texttt{MultTwenty} = \mathcal{H}_3$ of Fig.4. □

Because of Th. 1, the algorithm $\uplus$ is effective as soon as the underlying domain of observables is suitable for a static analysis, i.e. the abstract elements are computer representable, the order on P is decidable, and a widening operator ensures the convergence of the fixpoint computation. The dual operator, i.e., the elimination of a class from a hierarchy, corresponds straightforwardly to the algorithm for removing a node from an ordered tree [6].

5.3 Common Semantic Superclass

From the previous section we were left to define (and prove correct) the algorithm that returns the common *semantic* superclass (CSS) of two given classes. First we recall the definition of meaning-preserving transformation τ [12]:

Definition 7 (Program Transformation). *Let* $\texttt{A} = \langle \texttt{F}, \texttt{init}, \texttt{M} \rangle$ *and* $\langle \alpha, \gamma \rangle$ *a Galois connection satisfying (2). A meaning-preserving program transformation* $\tau \in [\texttt{F} \rightarrow \texttt{M} \rightarrow \texttt{M}]$ *is such that* $\forall \texttt{f} \in \texttt{F}.\ \forall \texttt{m} \in \texttt{M}$*: (i)* $\tau_{\texttt{f}}(\texttt{m})$ *does not contain the field* `f` *and (ii)* $\forall \bar{\texttt{d}} \in P.\ \alpha(\mathbb{M}^{>}[\![\texttt{m}]\!](\gamma(\bar{\texttt{d}})) \sqsubseteq \alpha(\mathbb{M}^{>}[\![\tau_{\texttt{f}}(\texttt{m})]\!](\gamma(\bar{\texttt{d}})))$.

Intuitively, $\tau_{\texttt{f}}(\texttt{m})$ projects out the field `f` from the source of `m` preserving the semantics up to an observation (i.e., α).

The algorithm CSS is presented in Fig. 7. It is parameterized by the underlying abstract domain of observables and a meaning preserving map τ. The algorithm starts with a superclass for `A` (i.e., $\langle \emptyset, \texttt{init}_{\texttt{A}}, \emptyset \rangle$). Then, it iterates by non-deterministically adding, at at each step, a field or a method of `A`: if such an addition produces a superclass for `B` then it is retained, otherwise it is discarded. When no more methods or fields can be added, the algorithm returns a semantic superclass for `A` and `B`, as guaranteed by the following theorem:

Theorem 5 (Soundness of CSS). *Let* `A` *and* `B` *be two classes. Then* CSS(`A`,`B`) *is such that* $\texttt{A} \lhd$ CSS(`A`,`B`) *and* $\texttt{B} \lhd$ CSS(`A`,`B`).

It is worth noting that in general, CSS(`A`,`B`) $\neq$ CSS(`B`,`A`). Furthermore, by Th. 1, it follows that if $\lhd$ is decidable, then the algorithm is effective. This is the case when the underlying abstract domain of observables corresponds to one used for a static analysis [20].

Example 9. Consider the classes `MultEight` and `MultTwelve` and `MultFour` defined as in Sect. 2. When using the abstract domain of linear congruences, CSS(`MultEight`,`MultTwelve`) = `MultFour`. □

5.4 Merging of Hierarchies

The last refactoring operation on hierarchies we consider is about merging. The algorithm $\uplus$ can be used as a basis for the algorithm to merge two admissible class hierarchies:

$$
\begin{array}{l}
\mathcal{H}_1 \Cup \mathcal{H}_2 \triangleq \textbf{let } \mathcal{H} = \mathcal{H}_1,\ N = \text{nodesOf}(\mathcal{H}_2) \\
\qquad\qquad \textbf{while } N \neq \emptyset \textbf{ do} \\
\qquad\qquad\quad \textbf{select } \mathtt{C} \in N \\
\qquad\qquad\quad \mathcal{H} = \mathcal{H} \uplus \mathtt{C},\ N = N - \mathtt{C} \\
\qquad\qquad \textbf{return } \mathcal{H}.
\end{array}
$$

Lemma 5. $\Cup$ *is a fair operator w.r.t.* $\lhd$, *i.e.,* $\Cup \in [\mathbb{H}[\lhd] \to \mathbb{H}[\lhd]]$.

It is worth mentioning that the modularity and modulability of the operators described in this section are the crucial keys that allow to apply them also to "*real world*" hierarchy management issues [33].

6 Related Work

In their seminal work on Simula [13], Dahl and Nygaard justified the concept of inheritance on syntactic bases, namely as textual concatenation of program blocks. A first semantic approach is [15] an (informal) operational approach to the semantics of inheritance is introduced. In particular the problem of specifying the semantics of message dispatch is reduced to that of method lookup. In [5] a denotational characterization of inheritance is introduced and proved correct w.r.t. an operational semantics based on the method lookup algorithm of [15]. An unifying view of the different forms of inheritance provided by programming languages is presented in [1]. In the *objects as records model* [2], the semantics of an object is abstracted with its type: inheritance is identified with subtyping. Such an approach is not fully satisfactory as shown in [4]. The notion of subtyping has been generalized in [18] where inheritance is seen as property preservation: the *behavioral type* of a class is a human-provided formula, which specifies the behavior of the class, and subclassing boils down to formula implication. The main difference between our concept of observable and that of behavioral type is that observables are systematically obtained as an abstraction of the class semantics instead of being provided by the programmer.

As for class hierarchies refactoring, [32] presents a semantics-preserving approach to class composition. Such an approach preserves the behavior of the composing hierarchies when they do not interfere. If they do interfere, a static analysis determines which components (classes, methods, etc.) of the hierarchies may interfere, given a set of programs that use such hierarchies. Such an approach is the base of the [33], which exploits static and dynamic information for class refactoring. The main difference between these works and ours is that we exploit the notion of observable, which is a property valid for *all* the instantiation contexts of a class. As a consequence we do not need to rely on a set of test programs for inferring hierarchy properties. Furthermore, as a soundness

requirement, we ask that a refactoring operator on a class hierarchy preserve the observable, i.e., an abstraction of the concrete semantics. As a consequence we are in a more general setting, and the traditional one is recovered as soon as we consider the domain of observables to be the concrete one.

7 Conclusions and Future Work

We introduced a framework for the definition and the manipulation of class hierarchies based on semantics abstraction. The main novelty of this approach is twofold: it provides a logic-based solid foundation of class refactoring operations that are safe by construction, and allows us to tune it according to the observed property. The next goal is the development of a tool for the semi-automatic refactoring of class hierarchies, based on [19, 21], and the design of abstract domains capturing properties expressible in JML [17].

Acknowledgments

Thanks to Radhia Cousot for her kind support. This work was partly supported by École Polytechnique, and by MIUR projects COFIN 2004013015 - FIRB RBAU018RCZ.

References

1. G. Bracha and W. R. Cook. Mixin-based inheritance. In *5th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '90)*, volume 25(10) of *SIGPLAN Notices*, pages 303–311, October 1990.
2. L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67, Berlin, 1984. Springer-Verlag. Full version in *Information and Computation*, 76(2/3):138–164, 1988.
3. G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.
4. W. R. Cook, W. Hill, and P. S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'90)*. ACM Press, January 1990.
5. W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, November 1994.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stei. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.
7. A. Cortesi and F. Logozzo. Abstract interpretation-based verification of non functional requirements. In *Proceedings of the 7th International Conference on Coordination Models and Languages (COORD'05)*, volume 3654 of *Lectures Notes in Computer Science*, pages 49–62. Springer-Verlag, April 2005.
8. P. Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 15, pages 843–993. Elsevier Science, 1990.

9. P. Cousot. Types as abstract interpretations, invited paper. In *24th ACM Symposium on Principles of Programming Languages (POPL '97)*, pages 316–331. ACM Press, January 1997.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, January 1977.
11. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.
12. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 178–190. ACM Press, New York, NY, January 2002.
13. O. Dahl and K. Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM (CACM)*, 9(9):671–678, September 1966.
14. D. L. Detlefs, K. Rustan M. Leino, G. Nelson, and Saxe J.B. Extended static checking. Research Report #159, Compaq Systems Research Center, Palo Alto, USA, December 1998.
15. A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
16. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT'91)*, volume 464 of *Lectures Notes in Computer Science*, pages 169–192. Springer-Verlag, April 1991.
17. G. T. Leavens, A. L. Baker, and C. Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*, November 2003.
18. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
19. F. Logozzo. Class-level modular analysis for object oriented languages. In *Proceedings of the 10th Static Analysis Symposium 2003 (SAS '03)*, volume 2694 of *Lectures Notes in Computer Science*, pages 37–54. Springer-Verlag, June 2003.
20. F. Logozzo. An approach to behavioral subtyping based on static analysis. In *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, April 2004.
21. F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, volume 2937 of *Lectures Notes in Computer Science*, pages 211–222. Springer-Verlag, January 2004.
22. F. Logozzo. *Modular Static Analysis of Object-oriented languges.* PhD thesis, École Polytecnique, 2004.
23. F. Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems and Structures*, 2005 (to appear).
24. F. Logozzo and A. Cortesi. Semantic class hierarchies by abstract intepretation. Technical Report CS-2004-7, Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy, 2004.
25. B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.
26. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

27. NetBeans.org and Sun Mycrosystem, Inc. Netbeans IDE, 2004.
28. P. Odifreddi. *Classical Recursion Theory*. Elsevier, Amsterdam, 1999.
29. J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, Chichester, 1994.
30. I. Pollet, B. Le Charlier, and A. Cortesi. Distinctness and sharing domains for static analysis of Java programs. In *Proceedings of the European Conference on Object Oriented Programming (ECOOP '01)*, volume 2072 of *Lectures Notes in Computer Science*, pages 77–98. Springer-Verlag, 2001.
31. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–288, 2002.
32. G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *Lectures Notes in Computer Science*, pages 562–584. Springer-Verlag, June 2002.
33. M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*. ACM Press, 2004.

Strong Preservation of Temporal Fixpoint-Based Operators by Abstract Interpretation

Francesco Ranzato and Francesco Tapparo

Dipartimento di Matematica Pura ed Applicata,
Università di Padova, Italy

Abstract. Standard abstract model checking relies on abstract Kripke structures which approximate the concrete model by gluing together indistinguishable states. Strong preservation for a specification language $\mathcal{L}$ encodes the equivalence of concrete and abstract model checking of formulas in $\mathcal{L}$. Abstract interpretation allows to design abstract models which are more general than abstract Kripke structures. In this paper we show how abstract interpretation-based models can be exploited in order to specify a general strongly preserving abstract model checking framework. This is shown in particular for specification languages including standard temporal operators which admit a characterization as least/greatest fixpoints, as e.g. standard "Finally", "Globally", "Until" and "Release" modalities.

1 Introduction

Abstract model checking is one successful and practical way to deal with the well-known state explosion problem of model checking in system verification [1, 3]. Standard abstract model checking [2] relies on abstract models which are based on partitions of the state space. Given a concrete model as a Kripke structure $\mathcal{K} = (\Sigma, \rightarrow)$, a standard abstract model is specified by an abstract Kripke structure $\mathcal{A} = (A, \rightarrow^\sharp)$ where the set A of abstract states is defined by a surjective map $h : \Sigma \rightarrow A$ and $\rightarrow^\sharp$ is an abstract transition relation on A. Thus, A determines a partition P_A of Σ and vice versa. A weak preservation result for some temporal language $\mathcal{L}$ guarantees that for any formula $\varphi \in \mathcal{L}$, if φ holds on the abstract model $\mathcal{A}$ then φ also holds on the concrete model $\mathcal{K}$. On the other hand, strong preservation means that any formula of $\mathcal{L}$ holds on $\mathcal{A}$ if and only if it holds on $\mathcal{K}$. Strong preservation is highly desirable since it allows to draw consequences from negative answers on the abstract side [3]. Thus, in order to design a standard abstract model we need both an appropriate partition of the space state and a suitable abstract transition relation.

The relationship between abstract interpretation and abstract model checking has been the subject of a number of works (see e.g. [2, 6, 7, 9, 10, 11, 15, 16, 19, 18]). We introduced in [17] an abstract interpretation-based framework for specifying generic strongly preserving abstract models, where a partition of the state space Σ is viewed as a particular abstract domain of the powerset $\wp(\Sigma)$, where $\wp(\Sigma)$ plays the role of concrete semantic domain. This generalized approach leads to a precise correspondence between forward complete abstract interpretations and strongly preserving abstract models. We deal with generic (temporal) languages $\mathcal{L}$ of state formulas which are inductively generated by a set AP of atomic propositions p and a set Op of operators f, i.e. $\mathcal{L} \ni \varphi ::= p \mid f(\varphi_1, ..., \varphi_n)$. A semantic interpretation $\boldsymbol{p} \subseteq \Sigma$ of atomic

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 332–347, 2006.
© Springer-Verlag Berlin Heidelberg 2006

propositions and of operators $\boldsymbol{f} : \wp(\Sigma)^n \to \wp(\Sigma)$ determines a concrete semantic function $[\![\cdot]\!] : \mathcal{L} \to \wp(\Sigma)$ where $[\![p]\!] = \boldsymbol{p}$ and $[\![f(\varphi_1, ..., \varphi_n)]\!] = \boldsymbol{f}([\![\varphi_1]\!], ..., [\![\varphi_n]\!])$. Thus, any abstract domain A of $\wp(\Sigma)$ and corresponding abstract interpretation $\boldsymbol{p}^\sharp \in A$ and $\boldsymbol{f}^\sharp : A^n \to A$ for constants/operators, denoted by $I^\sharp$, induce an abstract semantic function $[\![\cdot]\!]^A : \mathcal{L} \to A$ where $[\![p]\!]^A = \boldsymbol{p}^\sharp$ and $[\![f(\varphi_1, ..., \varphi_n)]\!]^A = \boldsymbol{f}^\sharp([\![\varphi_1]\!]^A, ..., [\![\varphi_n]\!]^A)$. In particular, the abstract interpretation of $\boldsymbol{p}$ and $\boldsymbol{f}$ can be given as their best correct approximations on A, i.e. $\boldsymbol{p}^A \stackrel{\text{def}}{=} \alpha(\boldsymbol{p})$ and $\boldsymbol{f}^A \stackrel{\text{def}}{=} \alpha \circ \boldsymbol{f} \circ \gamma$ where α and γ are the abstraction and concretization maps relating A to $\wp(\Sigma)$. In this generalized setting, strong preservation goes as follows: the abstract interpretation $(A, I^\sharp)$ is strongly preserving for $\mathcal{L}$ when for any $S \subseteq \Sigma$ and $\varphi \in \mathcal{L}$, $S \subseteq [\![\varphi]\!] \Leftrightarrow \alpha(S) \leq [\![\varphi]\!]^A$. When A is an abstract domain representing a partition of Σ, this boils down to standard strong preservation for abstract Kripke structures, where different choices for the abstract transition relation $\to^\sharp$ correspond to different abstract interpretations of the operators $\boldsymbol{f}$.

It turns out that forward completeness implies strong preservation, i.e. if the abstract domain A is forward complete for the concrete constants/operators of $\mathcal{L}$ — this means that no loss of precision occurs by approximating each $\boldsymbol{p}$ and $\boldsymbol{f}$ on the abstract domain A — then A is strongly preserving for $\mathcal{L}$. The converse is in general not true. However, we show that when A is $\mathcal{L}$-covered — meaning that each abstract value $a \in A$ corresponds to some formula $\varphi \in \mathcal{L}$, i.e. $\gamma(a) = [\![\varphi]\!]$ — forward completeness and strong preservation are indeed equivalent notions and consequently the abstract interpretation of constants/operators of $\mathcal{L}$ as best correct approximations on A is the only possible choice in order to have strong preservation. One interesting point to remark is that when the abstract domain is a state partition P, an abstract transition relation $\to^\sharp$ on P such that the abstract Kripke structure $(P, \to^\sharp)$ strongly preserves $\mathcal{L}$ might not exist, while, in contrast, a strongly preserving abstract semantics on the partition P viewed as an abstract domain always exists.

The abstract semantics is therefore defined by approximating the interpretation of logical/temporal operators of $\mathcal{L}$ through their best correct approximations on the abstract domain A. In principle, this can be done for any logical/temporal operator. However, when a temporal operator $\boldsymbol{f}$ can be expressed as a least/greatest fixpoint of another temporal operator $\boldsymbol{g}$, e.g. $\boldsymbol{f} = \lambda X. \operatorname{lfp}(\lambda Y.\boldsymbol{g}(X, Y))$, the best correct approximation $\alpha \circ \boldsymbol{f} \circ \gamma$ might not be characterizable as a least/greatest fixpoint. For example, the existential "Finally" operator can be characterized as a least fixpoint by $\mathbf{EF}(X) = \operatorname{lfp}(\lambda Y.X \cup \mathbf{EX}(Y))$, where $\mathbf{EX} = \operatorname{pre}_{\to}$ is the standard predecessor transformer on the concrete Kripke structure. The best correct approximation of $\mathbf{EF}$ on an abstract domain A is therefore the abstract function $\alpha \circ \mathbf{EF} \circ \gamma : A \to A$. However, this definition gives us no clue for computing $\alpha \circ \mathbf{EF} \circ \gamma$ as a least fixpoint. By contrast, in standard abstract model checking the abstract interpretation of language operators is based on an abstract Kripke structure $\mathcal{A} = (P, \to^\sharp)$, so that it is enough to compute the least fixpoint $\operatorname{lfp}(\lambda Y^\sharp.X^\sharp \cup \mathbf{EX}^\sharp(Y^\sharp))$ on the abstract state space P, namely $X^\sharp$ and $Y^\sharp$ are sets of blocks in P, $\cup$ is union of sets of blocks and $\mathbf{EX}^\sharp = \operatorname{pre}_{\to^\sharp}$ is the predecessor transformer on $\mathcal{A}$. For example, for the language $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EF}\varphi$ if one can define a strongly preserving abstract Kripke structure $(P, \to^\sharp)$, where P is some partition of Σ, then the abstract Kripke structure $(P, \to^{\exists\exists})$ strongly preserves $\mathcal{L}$ as well, where $B_1 \to^{\exists\exists} B_2$ iff $\exists s_1 \in B_1.\exists s_2 \in B_2. s_1 \to s_2$.

In this case, while the concrete fixpoint is given by $\mathbf{EF}(X) = \text{lfp}(\lambda Y.X \cup \text{pre}_{\rightarrow}(Y))$, the abstract fixpoint is $\mathbf{EX}^\sharp(X^\sharp) = \text{lfp}(\lambda Y^\sharp.X^\sharp \cup \text{pre}_{\rightarrow^{\exists\exists}}(Y^\sharp))$. The key point here is that the best correct approximation of the concrete function $\lambda\langle X, Y\rangle.\, X \cup \text{pre}_{\rightarrow}(Y)$ on the partition P viewed as an abstract domain is indeed $\lambda\langle X^\sharp, Y^\sharp\rangle.\, X^\sharp \cup \text{pre}_{\rightarrow^{\exists\exists}}(Y^\sharp)$. In other terms, the best correct approximation of $\lambda X.\, \text{lfp}(\lambda Y.X \cup \text{pre}_{\rightarrow}(Y))$ can be expressed as $\lambda X^\sharp.\, \text{lfp}(\lambda Y^\sharp.X^\sharp \cup \text{pre}_{\rightarrow^{\exists\exists}}(Y^\sharp))$ and thus preserves the same "template" of the concrete fixpoint function. We generalized this phenomenon to generic functions and abstract domains and then applied to standard temporal operators which can be expressed as fixpoints, that is, "Finally", "Globally", "Until" and "Release" modalities. We applied our results both to partitions, namely standard abstract models, and to disjunctive abstract domains, namely domains which are able to represent precisely logical disjunction. As far as partitions are concerned, we obtained new results of strong preservation on standard abstract Kripke structures. On the other hand, applications to disjunctive abstract domains provide a new procedure to perform a strongly preserving abstract model checking. This latter approach seems especially interesting because examples hint that efficient implementations are feasible.

2 Background

Notation. The standard pointwise ordering between functions will be denoted by $\sqsubseteq$. For a set $S \in \wp(\wp(X))$, we write the sets in S in a compact form like in $\{[1], [12], [123]\} \in \wp(\wp(\{1, 2, 3\}))$. We denote by $\complement$ the complement operator w.r.t. some universe set. $\text{Part}(\Sigma)$ denotes the set of partitions of Σ. We consider transition systems (Σ, R) where the relation $R \subseteq \Sigma \times \Sigma$ (also denoted by $\xrightarrow{R}$) is total. A Kripke structure $\mathcal{K} = (\Sigma, R, AP, \ell)$ consists of a transition system (Σ, R) together with a set AP of atomic propositions and a labelling function $\ell : \Sigma \rightarrow \wp(AP)$. Paths in $\mathcal{K}$ are defined by $\text{Path}(\mathcal{K}) \stackrel{\text{def}}{=} \{\pi : \mathbb{N} \rightarrow \Sigma \mid \forall i \in \mathbb{N}.\ \pi_i \xrightarrow{R} \pi_{i+1}\}$. A transition relation $R \subseteq \Sigma \times \Sigma$ defines the usual pre/post transformers on $\wp(\Sigma)$: pre_R, post_R, $\widetilde{\text{pre}}_R$, $\widetilde{\text{post}}_R$. When clear from the context, subscripts R are sometimes omitted. The relations $R^{\exists\exists}, R^{\forall\exists} \subseteq \wp(\Sigma) \times \wp(\Sigma)$ are defined as follows: $(S_1, S_2) \in R^{\exists\exists}$ (respectively, $R^{\forall\exists}$) iff $\exists s_1 \in S_1.$ (respectively, $\forall s_1 \in S_1.$) $\exists s_2 \in S_2.\, (s_1, s_2) \in R$.

Abstract Interpretation and Completeness. As usual in standard abstract interpretation, abstract domains are specified by Galois connections/insertions (GCs/GIs) [4, 5]. A GC/GI of the abstract domain A into the concrete domain C through the abstraction and concretization maps $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ will be denoted by (C, α, γ, A). GIs of a common concrete domain C are pre-ordered w.r.t. precision as usual: $\mathcal{G}_1 = (C, \alpha_1, \gamma_1, A_1) \sqsubseteq \mathcal{G}_2 = (C, \alpha_2, \gamma_2, A_2)$ (i.e., A_1 is more precise than A_2) iff $\gamma_1 \circ \alpha_1 \sqsubseteq \gamma_2 \circ \alpha_2$. Moreover, $\mathcal{G}_1$ and $\mathcal{G}_2$ are equivalent when $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ and $\mathcal{G}_2 \sqsubseteq \mathcal{G}_1$. Let $\mathcal{G} = (C, \alpha, \gamma, A)$ be a GI, $f : C \rightarrow C$ be some concrete semantic function — for simplicity, we consider here 1-ary functions — and $f^\sharp : A \rightarrow A$ be a corresponding abstract function. $\langle A, f^\sharp\rangle$ is a sound abstract interpretation when $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$. The abstract function $f^A \stackrel{\text{def}}{=} \alpha \circ f \circ \gamma : A \rightarrow A$ is called the best correct approximation of f in A. Completeness in abstract interpretation corresponds to require the following strengthening of soundness: $\alpha \circ f = f^\sharp \circ \alpha$. This is called *backward* completeness because an orthogonal notion of *forward* completeness may be considered: in fact, the

soundness condition $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ is equivalent to $f \circ \gamma \sqsubseteq \gamma \circ f^\sharp$, so that forward completeness for $f^\sharp$ corresponds to strengthen soundness by requiring: $f \circ \gamma = \gamma \circ f^\sharp$. Giacobazzi et al. [12] observed that both backward and forward completeness uniquely depend upon the abstraction map, namely they are abstract domain properties. In fact, it turns out that there exists $f^\sharp : A \to A$ such that $\langle A, f^\sharp \rangle$ is backward (forward) complete iff $\gamma \circ \alpha \circ f \circ \gamma \circ \alpha = \gamma \circ \alpha \circ f$ ($\gamma \circ \alpha \circ f \circ \gamma \circ \alpha = f \circ \gamma \circ \alpha$). Thus, we say that a GI $\mathcal{G}$ is backward (forward) complete for f when $\gamma \circ \alpha \circ f \circ \gamma \circ \alpha = \gamma \circ \alpha \circ f$ ($\gamma \circ \alpha \circ f \circ \gamma \circ \alpha = f \circ \gamma \circ \alpha$). Note that $\mathcal{G}$ is forward complete for f iff f maps elements in $\mathrm{img}(\gamma)$ to elements in $\mathrm{img}(\gamma)$.

If $[\![\cdot]\!] : \mathcal{L} \to C$ and $[\![\cdot]\!]^\sharp : \mathcal{L} \to A$ are, respectively, a concrete and an abstract semantics of a generic language $\mathcal{L}$, then soundness and completeness for the abstract semantics $[\![\cdot]\!]^\sharp$ are defined as follows: $\langle A, [\![\cdot]\!]^\sharp \rangle$ is sound (respectively, backward complete, forward complete) if for any $\varphi \in \mathcal{L}$, $\alpha([\![\varphi]\!]) \leq_A [\![\varphi]\!]^\sharp$ (respectively, $\alpha([\![\varphi]\!]) = [\![\varphi]\!]^\sharp$, $[\![\varphi]\!] = \gamma([\![\varphi]\!]^\sharp)$).

Recall that a GI $\mathcal{G} = (C, \alpha, \gamma, A)$ is disjunctive (or additive) when γ is additive, i.e. when γ preserves arbitrary least upper bounds. It turns out that $\mathcal{G}$ is disjunctive iff $\mathrm{img}(\gamma) \subseteq C$ is join-closed, i.e. closed under arbitrary lub's. Disjunctive GIs can be "inverted" as follows and such inversion preserves forward completeness.

Proposition 2.1. *Let $\mathcal{G} = (C_\leq, \alpha, \gamma, A_\leq)$ be a disjunctive GI and $f : C \to C$.*
(i) *Let $\alpha^\triangledown(c) \stackrel{\text{def}}{=} \vee \{a \in A \mid \gamma(a) \leq c\}$. Then, $\mathcal{G}^\triangledown \stackrel{\text{def}}{=} (C_\geq, \alpha^\triangledown, \gamma, A_\geq)$ is a GI.*
(ii) *$\mathcal{G}^\triangledown$ is forward complete for f iff $\mathcal{G}$ is forward complete for f. In this case, the two best correct approximations of f w.r.t. $\mathcal{G}^\triangledown$ and $\mathcal{G}$ coincide.*

3 Abstract Models

3.1 Abstract Semantics

We consider (temporal) specification languages $\mathcal{L}$ whose state formulas φ are inductively defined by: $\mathcal{L} \ni \varphi ::= p \mid f(\varphi_1, ..., \varphi_n)$, where $p \in AP$ ranges over a set of atomic propositions while f ranges over a finite set Op of operators. AP and Op are also denoted, respectively, by $AP_\mathcal{L}$ and $Op_\mathcal{L}$. Each $f \in Op$ has an arity $\mathrm{ar}(f) > 0$. The interpretation of formulas in $\mathcal{L}$ is determined by a *semantic structure* $\mathcal{S} = (\Sigma, I)$ where Σ is a set of states and I is an interpretation function which maps $p \in AP$ to $I(p) \in \wp(\Sigma)$ and $f \in Op$ to $I(f) : \wp(\Sigma)^{\mathrm{ar}(f)} \to \wp(\Sigma)$. We also use $\boldsymbol{p}$ and $\boldsymbol{f}$ to denote, respectively, $I(p)$ and $I(f)$. Also, $\boldsymbol{AP} \stackrel{\text{def}}{=} \{\boldsymbol{p} \in \wp(\Sigma) \mid p \in AP\}$ and $\boldsymbol{Op} \stackrel{\text{def}}{=} \{\boldsymbol{f} : \wp(\Sigma)^{\mathrm{ar}(f)} \to \wp(\Sigma) \mid f \in Op\}$. The *concrete state semantic function* $[\![\cdot]\!]_\mathcal{S} : \mathcal{L} \to \wp(\Sigma)$ evaluates a formula $\varphi \in \mathcal{L}$ to the set of states making φ true w.r.t. the semantic structure $\mathcal{S}$:

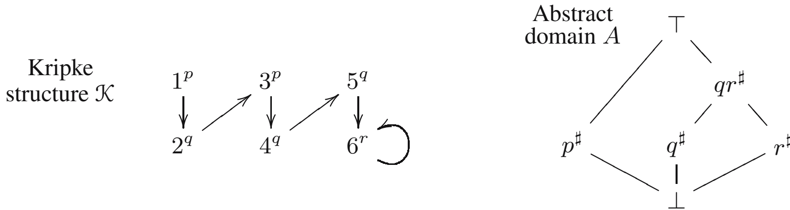
$$[\![p]\!]_\mathcal{S} = \boldsymbol{p} \quad \text{and} \quad [\![f(\varphi_1, ..., \varphi_n)]\!]_\mathcal{S} = \boldsymbol{f}([\![\varphi_1]\!]_\mathcal{S}, ..., [\![\varphi_n]\!]_\mathcal{S}).$$

Semantic structures generalize the role of Kripke structures. In fact, in standard model checking [3], a semantic structure is usually defined through a Kripke structure $\mathcal{K}$ so that the interpretation of operators in Op is defined in terms of paths in $\mathcal{K}$ and of standard logical operators. In the following, we will freely use standard logical and temporal operators together with their corresponding usual interpretations: for example, $I(\wedge) = \cap$, $I(\neg) = \complement$, $I(\mathrm{EX}) = \mathrm{pre}_R$, etc.

Following the abstract interpretation approach, an *abstract semantic structure* is given by $\mathcal{S}^\sharp = (A, I^\sharp)$ where (C, α, γ, A) is a GI and for any $p \in AP$ and $f \in Op$, $I(p) \in A$ and $I^\sharp(f) : A^{\mathrm{ar}(f)} \to A$. Thus, an abstract semantic structure $\mathcal{S}^\sharp$ defines an *abstract semantics* $[\![\cdot]\!]_{\mathcal{S}^\sharp} : \mathcal{L} \to A$ for the language $\mathcal{L}$.

Let $\mathcal{S}$ be a (concrete) semantic structure for $\mathcal{L}$. A GI (C, α, γ, A) always induces an abstract semantic structure $\mathcal{S}^A = (A, I^A)$ where I^A provides the best correct approximations on A of the concrete interpretation of constants/operators: $I^A(p) \stackrel{\mathrm{def}}{=} \alpha(I(p))$ for $p \in AP$ and $I^A(f) \stackrel{\mathrm{def}}{=} (I(f))^A$ for $f \in Op$. If the (concrete) interpretation $\boldsymbol{Op}_{\mathcal{L}}$ consists of monotone functions then the abstract semantics $[\![\cdot]\!]_{\mathcal{S}^A}$ induced by $\mathcal{S}^A$ is always automatically sound. This *induced abstract semantics* will be denoted by $[\![\cdot]\!]^A_{\mathcal{S}}$.

Example 3.1. Let us consider the following Kripke structure $\mathcal{K}$, where superscripts denote the labelling function.



Let $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \mathrm{EX}\varphi$. Let $\mathcal{S}$ be the semantic structure for $\mathcal{L}$ induced by the Kripke structure $\mathcal{K}$ so that $\mathbf{EX} = \mathrm{pre}_{\to}$. Let A be the lattice depicted above. We consider the abstraction map $\alpha : \wp(\Sigma)_\subseteq \to A$ where $\alpha(\{n\})$, i.e. on singletons, is defined by $\alpha(\{1\}) = \alpha(\{3\}) \stackrel{\mathrm{def}}{=} p^\sharp$, $\alpha(\{2\}) = \alpha(\{4\}) = \alpha(\{5\}) \stackrel{\mathrm{def}}{=} q^\sharp$ and $\alpha(\{6\}) \stackrel{\mathrm{def}}{=} r^\sharp$, while for any $S \in \wp(\Sigma)$, $\alpha(S) \stackrel{\mathrm{def}}{=} \vee_{s \in S}\, \alpha(\{s\})$. Hence, we have that:

$$[\![\mathrm{EX}r]\!]^A_{\mathcal{S}} = \mathbf{EX}^A([\![r]\!]^A_{\mathcal{S}}) = \mathbf{EX}^A(\alpha(\boldsymbol{r})) = \mathbf{EX}^A(\alpha(\{6\})) = \mathbf{EX}^A(r^\sharp) =$$
$$\alpha(\mathbf{EX}(\gamma(r^\sharp))) = \alpha(\mathbf{EX}(\{6\})) = \alpha(\{5,6\}) = \alpha(\{5\}) \vee \alpha(\{6\}) = q^\sharp \vee r^\sharp = qr^\sharp.$$

Since $\gamma(qr^\sharp) = \{2,4,5,6\}$, as expected, observe that the abstract semantics $[\![\mathrm{EX}r]\!]^A_{\mathcal{S}}$ is a proper over-approximation in A of the concrete semantics $[\![\mathrm{EX}r]\!]_{\mathcal{S}} = \{5,6\}$. □

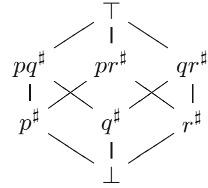
3.2 Partitioning Abstractions

As shown in [17], standard partition-based abstract model checking [2, 3] can be viewed as a particular instance of abstract semantics as defined in Section 3.1, where: (i) given some state partition $P \in \mathrm{Part}(\Sigma)$, the abstract domain is $\wp(P)_\subseteq$, where the abstraction map is the "covering" function $\alpha_P : \wp(\Sigma)_\subseteq \to \wp(P)_\subseteq$ such that $\alpha_P(S) \stackrel{\mathrm{def}}{=} \{B \in P \mid B \cap S \neq \varnothing\}$, while $\gamma_P : \wp(P)_\subseteq \to \wp(\Sigma)_\subseteq$ is given by $\gamma_P(X) = \cup_{B \in X} B$; (ii) if the concrete interpretation function I is based on a concrete Krike structure $\mathcal{K}$, then the abstract interpretation function $I^\sharp$ is simply given by the evaluation of I on an abstract Kripke structure $\mathcal{A} = (P, R^\sharp, AP, \ell^\sharp)$ which replaces $\mathcal{K}$, where $R^\sharp \subseteq P \times P$ is the abstract transition relation on the abstract state space P. Thus, in this sense, an abstract Kripke structure always induces an abstract semantics for a language.

Any GI $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ which is equivalent to a GI $(\wp(\Sigma)_\subseteq, \alpha_P, \gamma_P, \wp(P)_\subseteq)$, for some partition $P \in \mathrm{Part}(\Sigma)$, is called *partitioning*. It turns out (see [17]) that $\mathcal{G}$

is partitioning iff $\gamma(A)$ is closed under complementation. Of course, not every abstraction of $\wp(\Sigma)_{\subseteq}$ is partitioning. For instance, if $\bar{s} \in \Sigma$, $A = \{\bot, \top\}$, $\gamma(\bot) = \{\bar{s}\}$ and $\gamma(\top) = \Sigma$ then $(\wp(\Sigma)_{\subseteq}, \alpha, \gamma, A)$ is a disjunctive GI, where α denotes the left adjoint to γ, which is not partitioning because $\gamma(A) = \{\{\bar{s}\}, \Sigma\}$ is not closed under complementation. This opens the question whether it is possible to minimally refine a given abstract domain in order to make it partitioning. Given a GI $\mathcal{G} = (\wp(\Sigma)_{\subseteq}, \alpha, \gamma, A)$, we define an equivalence relation $\sim_{\mathcal{G}}$ on Σ by identifying those states that are blurred by the abstraction α: $s \sim_{\mathcal{G}} t$ iff $\alpha(\{s\}) = \alpha(\{t\})$. This is an equivalence relation, namely a partition in $\mathrm{Part}(\Sigma)$, and therefore it induces a partitioning abstraction that we denote by $\mathbb{P}(\mathcal{G})$. As shown in [17], it turns out that $\mathbb{P}(\mathcal{G})$ is the least partitioning refinement of $\mathcal{G}$, that is: $\mathbb{P}(\mathcal{G}) \sqsubseteq \mathcal{G}$ and for any partitioning $\mathcal{G}' \sqsubseteq \mathcal{G}$, $\mathcal{G}' \sqsubseteq \mathbb{P}(\mathcal{G})$.

Example 3.2. Let us consider the abstraction $\mathcal{G}$ in Example 3.1. From the definition of α, we have that $\alpha(\{s\}) = \alpha(\{t\})$ iff s and t belong to the same block of the partition $P = \{[13], [245], [6]\}$, so that $\mathbb{P}(\mathcal{G})$ is given by the GI $(\wp(\Sigma), \alpha_P, \gamma_P, \wp(P))$. The abstract domain $\wp(P)_{\subseteq}$ can be therefore represented by the lattice depicted on the right. □



3.3 Strong Preservation

As recalled above, standard abstract model checking [2, 3] is based on state partitions and abstract Kripke structures. Strong preservation for some language $\mathcal{L}$ encodes the equivalence of abstract and concrete validity for formulas in $\mathcal{L}$. Given a partition $P \in \mathrm{Part}(\Sigma)$, let $[\![\cdot]\!]^P : \mathcal{L} \to \wp(P)$ denote an abstract semantics defined on $\wp(P)$. For example, but not necessarily, this can be the abstract semantics induced by an abstract Kripke structure $(P, R^\sharp, AP, \ell^\sharp)$. A partition $P \in \mathrm{Part}(\Sigma)$ is *strongly preserving* (s.p. for short) for $\mathcal{L}$ when for any $s \in \Sigma$ and $\varphi \in \mathcal{L}$, $s \in [\![\varphi]\!]$ iff $\alpha_P(\{s\}) \in [\![\varphi]\!]^P$. It is known [8, 9, 17] that the coarsest s.p. partition $P_{\mathcal{L}}$ for $\mathcal{L}$ is given by the following state equivalence $\sim_{\mathcal{L}}$ induced by $\mathcal{L}$: $s_1 \sim_{\mathcal{L}} s_2$ iff $\forall \varphi \in \mathcal{L}.\ s_1 \in [\![\varphi]\!] \Leftrightarrow s_2 \in [\![\varphi]\!]$. Obviously, the definition of an abstract Kripke structure which induces a s.p. abstract semantics depends on the language $\mathcal{L}$. Let us recall some well-known examples [2, 3, 13]. Let $\mathcal{K} = (\Sigma, R, AP, \ell)$ be a concrete Kripke structure and let $P_{\mathrm{sim}}, P_{\mathrm{bis}} \in \mathrm{Part}(\Sigma)$ denote, respectively, simulation and bisimulation equivalence on $\mathcal{K}$. Then, the abstract semantics induced by the abstract Kripke structure $(P_{\mathrm{sim}}, R^{\forall\exists}, AP, \ell^\sharp)$ (where $\ell^\sharp(B) = \cup_{s \in B} \ell(s)$) is s.p. for ACTL*, while that induced by $(P_{\mathrm{bis}}, R^{\exists\exists}, AP, \ell^\sharp)$ is s.p. for CTL*.

Strong preservation was generalized in [17] to abstract domains as follows.

Definition 3.3. Let $\mathcal{S} = (\Sigma, I)$ and $\mathcal{S}^\sharp = (A, I^\sharp)$ be, respectively, concrete and abstract semantic structures for $\mathcal{L}$. Let $[\![\cdot]\!]_{\mathcal{S}^\sharp} : \mathcal{L} \to A$ be the corresponding abstract semantics. $\mathcal{S}^\sharp$ (or $[\![\cdot]\!]_{\mathcal{S}^\sharp}$) is *strongly preserving* for $\mathcal{L}$ (w.r.t. $\mathcal{S}$) when for any $S \in \wp(\Sigma)$ and $\varphi \in \mathcal{L}$, $S \subseteq [\![\varphi]\!]_{\mathcal{S}} \Leftrightarrow \alpha(S) \leq_A [\![\varphi]\!]_{\mathcal{S}^\sharp}$. □

The following simple but key result shows that strong preservation amounts to forward completeness.

Theorem 3.4. *$\mathcal{S}^\sharp$ is s.p. for $\mathcal{L}$ iff the abstract semantics $\langle A, [\![\cdot]\!]_{\mathcal{S}^\sharp}\rangle$ is forward complete.*

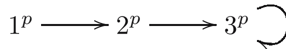
It turns out (cf. [17]) that if a s.p. abstract semantics on the abstract domain A exists then the abstract semantics $[\![\cdot]\!]^A_{\mathcal{S}}$ induced by A is s.p. as well, so that strong preservation is an *abstract domain property*. Hence, we say that the GI $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ (or simply A when the GI is clear from the context) is s.p. for $\mathcal{L}$ if $\mathcal{S}^A$ is s.p. for $\mathcal{L}$ (or, equivalently, if a s.p. abstract semantics on A exists). In this case, by Theorem 3.4, we also say that the abstract domain A is *language forward complete* for $\mathcal{L}$.

Example 3.5. Let us consider again Example 3.1. It turns out that A is not s.p. preserving for $\mathcal{L}$ because $\gamma([\![\mathrm{EX}r]\!]^A_{\mathcal{S}}) = \gamma(qr^\sharp) = \{2,4,5,6\}$, while $[\![\mathrm{EX}r]\!]_{\mathcal{S}} = \{5,6\}$. Therefore, for instance, $2 \in \gamma([\![\mathrm{EX}p]\!]^A_{\mathcal{S}}) \smallsetminus [\![\mathrm{EX}r]\!]_{\mathcal{S}}$, or, equivalently, $\alpha(\{2\}) \leq [\![\mathrm{EX}r]\!]^A_{\mathcal{S}}$ whilst $2 \notin [\![\mathrm{EX}r]\!]_{\mathcal{S}}$. □

4 Abstract Semantics

It is known (see e.g. [7]) that if an abstract domain A is forward complete for all the constants/operators of $\boldsymbol{AP} \cup \boldsymbol{Op}$ (where atomic propositions are viewed as 0-ary operators) — here also called *operator-wise forward completeness* — of some concrete interpretation of some language $\mathcal{L}$ then A is language forward complete for $\mathcal{L}$ (i.e., for all $\varphi \in \mathcal{L}$, $[\![\varphi]\!]_{\mathcal{S}} = \gamma([\![\varphi]\!]^A_{\mathcal{S}})$). The converse in general is not true, as shown by the following example.

Example 4.1. Let us consider the following Kripke structure $\mathcal{K}$ and the partitioning abstract domain A induced by the partition $P = \{[12], [3]\}$, i.e. $A = \wp(P)_\subseteq$.



Let us consider the language $\mathcal{L} \ni \varphi ::= p \mid \mathrm{EX}\varphi$. The Kripke structure $\mathcal{K}$ induces the semantic structure $\mathcal{S} = (\{1,2,3\}, I)$ such that $I(p) = \{1,2,3\}$ and $I(\mathrm{EX}) = \mathrm{pre}_\rightarrow$. Hence, we have that $[\![p]\!]_{\mathcal{S}} = \{1,2,3\}$, $[\![\mathrm{EX}p]\!]_{\mathcal{S}} = \{1,2,3\}$ and, for $k > 1$, $[\![\mathrm{EX}^k p]\!]_{\mathcal{S}} = \{1,2,3\}$. On the abstract side we have that $[\![p]\!]^A_{\mathcal{S}} = \{[12],[3]\}$, $[\![\mathrm{EX}p]\!]^A_{\mathcal{S}} = \{[12],[3]\}$ and, for $k > 1$, $[\![\mathrm{EX}^k p]\!]^A_{\mathcal{S}} = \{[12],[3]\}$. Thus, for any $\varphi \in \mathcal{L}$, $[\![\varphi]\!]_{\mathcal{S}} = \gamma_P([\![\varphi]\!]^A_{\mathcal{S}})$, i.e. the abstract domain A is language forward complete for $\mathcal{L}$. On the other hand, $\mathrm{pre}_\rightarrow(\gamma_P(\alpha_P(\{3\}))) = \mathrm{pre}_\rightarrow(\{3\}) = \{2,3\}$ while $\gamma_P(\alpha_P(\mathrm{pre}_\rightarrow(\gamma_P(\alpha_P(\{3\}))))) = \gamma_P(\alpha_P(\{2,3\})) = \{1,2,3\}$, so that A is not forward complete for $\mathrm{pre}_\rightarrow$. □

Operator-wise forward completeness is easier to check than language forward completeness. Moreover, the problem of refining an abstract domain in order to make it forward (or backward) complete for a given set of operators admits constructive fixpoint solutions [12, 18]. It is thus interesting to determine conditions on abstract domains which guarantee the equivalence of operator-wise and language forward completeness.

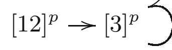
Definition 4.2. Let $\mathcal{S} = (\Sigma, I)$ be a semantic structure for $\mathcal{L}$ and $(\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ be a GI. The abstract domain A is $\mathcal{L}$-*covered* by the concrete semantics $[\![\cdot]\!]_{\mathcal{S}}$ (or simply by $\mathcal{S}$) when for any $a \in A$ there exists $\varphi \in \mathcal{L}$ such that $\gamma(a) = [\![\varphi]\!]_{\mathcal{S}}$. □

It turns out that this notion of covering ensures the equivalence of operator-wise and language forward completeness.

Theorem 4.3. *Let A be $\mathcal{L}$-covered by $\mathcal{S}$. Then, A is language forward complete for $\mathcal{L}$ iff A is forward complete for all the constants/operators in $\boldsymbol{AP}_{\mathcal{L}} \cup \boldsymbol{Op}_{\mathcal{L}}$.*

As recalled above, given an abstract domain A, if an abstract semantic structure $\mathcal{S}^\sharp = (A, I^\sharp)$ is s.p. for $\mathcal{L}$ then the abstract structure $\mathcal{S}^A = (A, I^A)$ induced by A is s.p. for $\mathcal{L}$ as well. However, the interpretation functions $I^\sharp$ and I^A may differ.

Example 4.4. Let us consider again Example 4.1. Let us first note that A is not $\mathcal{L}$-covered by $\mathcal{S}$ because $\{[\![\varphi]\!]_{\mathcal{S}} \mid \varphi \in \mathcal{L}\} = \{\{1,2,3\}\}$. Let us consider the abstract semantic structure $\mathcal{S}^\sharp = (A, I^\sharp)$ induced by the following abstract Kripke structure:



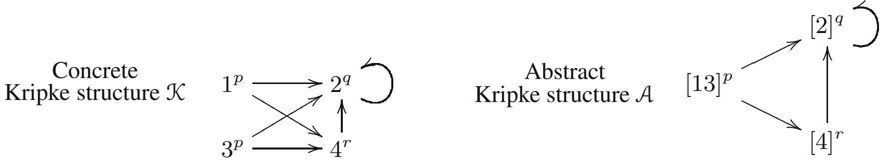
Hence, $I^\sharp(\mathrm{EX}) = \mathrm{pre}_{R^\sharp}$ where $\mathrm{pre}_{R^\sharp}(\varnothing) = \varnothing$, $\mathrm{pre}_{R^\sharp}(\{[12]\}) = \varnothing$, $\mathrm{pre}_{R^\sharp}(\{[3]\}) = \{[12],[3]\}$, $\mathrm{pre}_{R^\sharp}(\{[12],[3]\}) = \{[12],[3]\}$. It is easy to see that $\mathcal{S}^\sharp$ is s.p. for $\mathcal{L}$. In fact, we have that $\gamma_P([\![p]\!]_{\mathcal{S}^\sharp}) = \gamma_P(\{[12],[3]\}) = \{1,2,3\} = [\![p]\!]_{\mathcal{S}}$ and $\gamma_P([\![\mathrm{EX}p]\!]_{\mathcal{S}^\sharp}) = \gamma_P(\mathrm{pre}_{R^\sharp}(\{[12],[3]\})) = \gamma_P(\{[12],[3]\}) = \{1,2,3\} = [\![\mathrm{EX}p]\!]_{\mathcal{S}}$, so that by Theorem 3.4, $\mathcal{S}^\sharp$ is s.p. for $\mathcal{L}$. However, it turns out that $I^\sharp(\mathrm{EX}) \neq I^A(\mathrm{EX}) = \alpha_P \circ \mathrm{pre}_{\to} \circ \gamma_P$. In fact, $\mathrm{pre}_{R^\sharp}(\{[12]\}) = \varnothing$ while $\alpha_P(\mathrm{pre}_{\to}(\gamma_P(\{[12]\}))) = \alpha_P(\mathrm{pre}_{\to}(\{1,2\})) = \alpha_P(\{1\}) = \{[12]\}$. Thus, $\mathcal{S}^\sharp$ and $\mathcal{S}^A$ are two different abstract semantic structures which are both s.p. for $\mathcal{L}$. □

Thus, in general, for a given abstract domain A, there may be different s.p. abstract semantic structures defined over A. However, if A is $\mathcal{L}$-covered by the concrete semantic structure then a unique s.p. abstract semantic structure may exist.

Corollary 4.5. *If A is $\mathcal{L}$-covered by $\mathcal{S}$ and $\mathcal{S}^\sharp = (A, I^\sharp)$ is s.p. for $\mathcal{L}$ then $I^\sharp = I^A$.*

Thus, when A is $\mathcal{L}$-covered by $\mathcal{S}$, we have that a *unique interpretation* of constants/functions on A which is s.p. for $\mathcal{L}$ may exist, namely their best correct approximations on A.

Example 4.6. Let us consider the language $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EX}\varphi$ and the following Kripke structure $\mathcal{K}$ with transition relation R.



This induces a concrete semantic structure $\mathcal{S} = (\{1,2,3,4\}, I)$ where $I(p) = \{1,3\}$, $I(q) = \{2\}$, $I(r) = \{4\}$, $I(\neg) = \complement$, $I(\wedge) = \cap$ and $I(\mathrm{EX}) = \mathrm{pre}_R$. Let us consider the state partition $P = \{13, 2, 4\}$ and the corresponding abstract Kripke structure $\mathcal{A}$ depicted above where the transition relation is given by $R^{\exists\exists}$. Let us consider the abstract semantic structure $\mathcal{S}^\sharp = (A, I^\sharp)$ induced by $\mathcal{A}$, i.e. $A = \wp(P)_{\subseteq}$ and $I^\sharp(p) = \{13\}$, $I^\sharp(q) = \{2\}$, $I^\sharp(r) = \{4\}$, $I^\sharp(\neg) = \complement$, $I^\sharp(\wedge) = \cap$ and $I^\sharp(\mathrm{EX}) = \mathrm{pre}_{R^{\exists\exists}}$.

It is easy to check that $I^\sharp(\neg)$, $I^\sharp(\wedge)$ and $I^\sharp(\mathrm{EX})$ are indeed the best correct approximations on A of, respectively, the concrete operations of set complementation $\complement = I(\neg)$, set interesection $\cap = I(\wedge)$ and $\mathrm{pre}_R = I(\mathrm{EX})$. Hence, $I^\sharp = I^A$, namely $\mathcal{S}^\sharp = \mathcal{S}^A$.

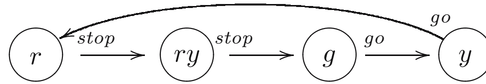
It turns out that A is $\mathcal{L}$-covered by $\mathcal{S}$. In fact, since the set of concrete semantics of formulas in $\mathcal{L}$ is closed under set complementation we have that any union of blocks of P belongs to $\{[\![\varphi]\!]_{\mathcal{S}} \mid \varphi \in \mathcal{L}\}$, so that $\mathrm{img}(\gamma_P) \subseteq \{[\![\varphi]\!]_{\mathcal{S}} \mid \varphi \in \mathcal{L}\}$.

We also have that $\mathcal{S}^A$ is s.p. for $\mathcal{L}$. This happens because A is forward complete for the constants/operations of $\mathcal{L}$. In fact, all the concrete operations $\complement$, $\cap$ and pre_R map unions of blocks in $\wp(P)$ into unions of blocks in $\wp(P)$ and therefore the abstract domain $A = \wp(P)$ is forward complete for them. For example, let us observe that this holds for pre_R because $\mathrm{pre}_R(\{1,3\}) = \varnothing$, $\mathrm{pre}_R(\{2\}) = \{1,3,4\}$ and $\mathrm{pre}_R(\{4\}) = \{1,3\}$. Hence, since A is operator-wise forward complete we have that A is language forward complete for $\mathcal{L}$ as well and therefore, by Theorem 3.4, $\mathcal{S}^A$ is s.p. for $\mathcal{L}$.

Consequently, by Corollary 4.5, $\mathcal{S}^A$ is the unique abstract semantic structure on the abstract domain A which is s.p. for $\mathcal{L}$. □

It may also happen that one can define a s.p. abstract semantics on some partition P although this abstract semantics cannot be derived from an abstract Kripke structure on P, as shown by the following example.

Example 4.7. Consider the following simple language $\mathcal{L} \ni \varphi ::= p \mid \mathrm{AXX}\varphi$ and the following Kripke structure $\mathcal{K}$ where R is the transition relation.



This models a four-state traffic light controller (like in the U.K. and in Germany). This gives rise to a concrete semantic structure $\mathcal{S} = (\{r, ry, g, y\}, I)$ where $I(\mathit{stop}) = \{r, ry\}$, $I(\mathit{go}) = \{g, y\}$ and $I(\mathrm{AXX}) = \widetilde{\mathrm{pre}}_{R^2}$. Hence, according to the standard interpretation $I(\mathrm{AXX}) = \widetilde{\mathrm{pre}}_{R^2}$, we have that $s \in [\![\mathrm{AXX}\varphi]\!]_{\mathcal{S}}$ iff for any path $\pi_0\pi_1\pi_2\ldots$ in $\mathcal{K}$ starting from $s = \pi_0$, we have that $\pi_2 \in [\![\varphi]\!]_{\mathcal{S}}$. Observe that $[\![\mathrm{AXX}\mathit{stop}]\!]_{\mathcal{S}} = \{g, y\}$ and $[\![\mathrm{AXX}\mathit{go}]\!]_{\mathcal{S}} = \{r, ry\}$. Consider the partition $P = \{[r, ry], [g, y]\}$ and the corresponding partitioning abstract domain $A = \wp(P)_{\subseteq}$. Hence, for the corresponding abstract semantic structure $\mathcal{S}^A = (A, I^A)$ we have that $I^A(\mathit{stop}) = \{[r, ry]\}$, $I^A(\mathit{go}) = \{[g, y]\}$ and $I^A(\mathrm{AXX}) = \alpha_P \circ \widetilde{\mathrm{pre}}_{R^2} \circ \gamma_P$, so that

$$
\begin{aligned}
&I^A(\mathrm{AXX})(\varnothing) = \varnothing;\\
&I^A(\mathrm{AXX})(\{[r, ry]\}) = \{[g, y]\}; \quad I^A(\mathrm{AXX})(\{[g, y]\}) = \{[r, ry]\};\\
&I^A(\mathrm{AXX})(\{[r, ry], [g, y]\}) = \{[r, ry], [g, y]\}.
\end{aligned}
$$

By Theorem 3.4, it turns out that $\mathcal{S}^A$ is s.p. for $\mathcal{L}$ because A is forward complete for $\widetilde{\mathrm{pre}}_{R^2}$. In fact, it turns out that $\widetilde{\mathrm{pre}}_{R^2}$ maps unions of blocks in P to unions of blocks in P because: $\widetilde{\mathrm{pre}}_{R^2}(\varnothing) = \varnothing$, $\widetilde{\mathrm{pre}}_{R^2}(\{r, ry\}) = \{g, y\}$, $\widetilde{\mathrm{pre}}_{R^2}(\{g, y\}) = \{r, ry\}$ and $\widetilde{\mathrm{pre}}_{R^2}(\{r, ry, g, y\}) = \{r, ry, g, y\}$.

However, let us show that there exists no abstract transition relation $R^\sharp \subseteq P \times P$ on the partition P such that the abstract Kripke structure $\mathcal{A} = (P, R^\sharp, AP, \ell^\sharp)$ induces an abstract semantic structure which is s.p. for $\mathcal{L}$. Assume by contradiction that such an abstract Kripke structure $\mathcal{A}$ exists and let $\mathcal{S}^\sharp$ be the corresponding induced abstract semantic structure. Let $B_1 = [r, ry] \in P$ and $B_2 = [g, y] \in P$. Since $r \in [\![\mathrm{AXX}\mathit{go}]\!]_{\mathcal{S}}$ and $g \in [\![\mathrm{AXX}\mathit{stop}]\!]_{\mathcal{S}}$, by strong preservation, it must be that $B_1 \in [\![\mathrm{AXX}\mathit{go}]\!]_{\mathcal{S}^\sharp}$ and $B_2 \in [\![\mathrm{AXX}\mathit{stop}]\!]_{\mathcal{S}^\sharp}$. Thus, necessarily, $(B_1, B_2), (B_2, B_1) \in R^\sharp$. This leads to the

contradiction $B_1 \not\in [\![\mathrm{AXX}go]\!]_{\mathbb{S}^\sharp}$. In fact, if $R^\sharp = \{(B_1, B_2), (B_2, B_1)\}$ then we would have that $B_1 \not\in [\![\mathrm{AXX}go]\!]_{\mathbb{S}^\sharp}$. Moreover, if, instead, $(B_1, B_1) \in R^\sharp$ (the case (B_2, B_2) is analogous), then we would still have that $B_1 \not\in [\![\mathrm{AXX}go]\!]_{\mathbb{S}^\sharp}$. Even more, along the same lines it is not difficult to show that no proper abstract Kripke structure induces an abstract semantic structure which strongly preserves $\mathcal{L}$, because even if we split one of the two blocks B_1 or B_2 we still cannot define an abstract transition relation ensuring strong preservation for $\mathcal{L}$. □

5 Fixpoints in Abstract Semantics

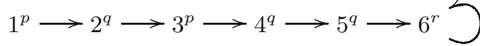
The above abstract interpretation-based approach to abstract model checking systematically defines the abstract semantics by approximating the interpretation of logical/temporal operators through their best correct approximations on the abstract domain. In principle, this can be done for any logical/temporal operator. However, when a temporal operator $\boldsymbol{f}$ can be expressed as a least/greatest fixpoint of another temporal operator $\boldsymbol{g}$, e.g. $\boldsymbol{f}(S) = \mathrm{lfp}(\lambda X.\boldsymbol{g}(X, S))$, the best correct approximation $\alpha \circ \boldsymbol{f} \circ \gamma$ might not be characterizable as a least/greatest fixpoint. Ideally, we would aim at approximating $\boldsymbol{g}$ through some abstract operator $\boldsymbol{g}^\sharp$ in order to be able to characterize $\alpha \circ \boldsymbol{f} \circ \gamma$ as the abstract least fixpoint of $\boldsymbol{g}^\sharp$. Let us illustrate this through the case of the "Finally" operator EF, whose standard interpretation can be characterized as a fixpoint: $\mathbf{EF}(S) = \mathrm{lfp}(\lambda X.S \cup \mathbf{EX}(X))$. The best correct approximation of $\mathbf{EF}$ w.r.t. a Galois insertion $(\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ is the abstract function $\alpha \circ \mathbf{EF} \circ \gamma : A \to A$. However, this definition gives us no clue for computing $\alpha \circ \mathbf{EF} \circ \gamma$ as a least fixpoint. By contrast, in standard abstract model checking the abstract interpretation of the language operators is based on an abstract transition relation defined on the abstract state space, i.e. an abstract Kripke structure, so that it is enough to compute the least fixpoint $\mathrm{lfp}(\lambda X.S \cup \mathbf{EX}(X))$ on the abstract Kripke structure. For example, consider the language $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EF}\varphi$. Let $\mathcal{K} = (\Sigma, R, AP, \ell)$ be a concrete Kripke structure. One can easily see that if $P \in \mathrm{Part}(\Sigma)$ is s.p. for $\mathcal{L}$ then the abstract Kripke structure on P with abstract transition relation $R^{\exists\exists} \subseteq P \times P$ is s.p. for $\mathcal{L}$. In this case, while the concrete fixpoint is given by $\mathbf{EF}(S) = \mathrm{lfp}(\lambda X.S \cup \mathrm{pre}_R(X))$, for any $S \subseteq \Sigma$, the abstract fixpoint is $\mathrm{lfp}(\lambda X^\sharp.S^\sharp \cup_P \mathrm{pre}_{R^{\exists\exists}}(X^\sharp))$, for any $S^\sharp \subseteq P$, where $\cup_P$ is union of blocks in P, namely the least upper bound of the abstract domain $\wp(P)_\subseteq$. Recall that the abstract domain $\wp(P)_\subseteq$ is related to the concrete domain $\wp(\Sigma)_\subseteq$ by the GI $\mathcal{G}_P = (\wp(\Sigma)_\subseteq, \alpha_P, \gamma_P, \wp(P)_\subseteq)$. The key point to note here is that $\lambda\langle X^\sharp, Y^\sharp\rangle.\, X^\sharp \cup^A \mathrm{pre}^A_R(Y^\sharp)$ is indeed the best correct approximation of the concrete operation $\lambda\langle X, Y\rangle.\, X \cup \mathrm{pre}_R(Y)$ through the GI $\mathcal{G}_P$. These observations lead us to the following generalization.

Theorem 5.1. *Let C be a complete lattice, (C, α, γ, A) be a GI and $f : C^{n+1} \to C$ be monotone. Let $F \stackrel{\text{def}}{=} \lambda\vec{c} \in C^n.\, \mathrm{lfp}(\lambda x.f(c_1, ..., x, ..., c_n))$. If A is forward complete for F then $F^A = \lambda\vec{a} \in A^n.\mathrm{lfp}(\lambda x.f^A(a_1, ..., x, ..., a_n))$.*

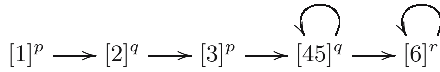
Let us remark that the above result can be also stated by duality for greatest fixpoints as follows: if $(C_\geq, \alpha, \gamma, A_\geq)$ is a GI, $F \stackrel{\text{def}}{=} \lambda\vec{c} \in C^n.\, \mathrm{gfp}(\lambda x.f(c_1, ..., x, ..., c_n))$ and A is forward complete for F then $F^A = \lambda\vec{a} \in A^n.\mathrm{gfp}(\lambda x.f^A(a_1, ..., x, ..., a_n))$.

By Theorems 3.4 and 4.3, given a language $\mathcal{L}$ and a semantic structure $\mathcal{S}$ for $\mathcal{L}$, if A is $\mathcal{L}$-covered by $\mathcal{S}$ then A is forward complete for the constants/operators in $\boldsymbol{AP}_{\mathcal{L}} \cup \boldsymbol{Op}_{\mathcal{L}}$ iff $\mathcal{S}^A$ is s.p. for $\mathcal{L}$. Thus, in this case, by Theorem 5.1, if $\mathcal{S}^A$ is s.p. for $\mathcal{L}$ and $\boldsymbol{Op}_{\mathcal{L}}$ includes an operator $\boldsymbol{f}$ which can be expressed as a least/greatest fixpoint of some operation $\boldsymbol{g}$ then the best correct approximation of $\boldsymbol{f}$ on A can be obtained as the abstract least/greatest fixpoint of the best correct approximation of $\boldsymbol{g}$ on A.

Example 5.2. Let us consider $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EF}\varphi$ and the following Kripke structure $\mathcal{K}$ with transition relation R which induces a concrete semantic structure $\mathcal{S}$.



Let us consider the partition $P = \{[1], [2], [3], [45], [6]\}$ and the corresponding abstract Kripke structure $\mathcal{A}$ depicted below where the transition relation is given by $R^{\exists\exists}$.



Let $\mathcal{S}^A$ be the abstract semantic structure induced by the abstract domain $A = \wp(P)_{\subseteq}$. It turns out that $\mathcal{S}^A$ is s.p. for $\mathcal{L}$ because A is forward complete for ($\boldsymbol{AP}_{\mathcal{L}}$ and) $\boldsymbol{Op}_{\mathcal{L}} = \{\cap, \complement, \mathbf{EF}\}$: in fact, it is easy to check that A is forward complete for $\mathbf{EF}$ because $\mathbf{EF}$ maps unions of blocks in P to unions of blocks in P. Since A is forward complete for $\mathbf{EF}$ and $\mathbf{EF}(S) = \mathrm{lfp}(\lambda X.f(S, X))$, where $f(S, X) \stackrel{\text{def}}{=} S \cup \mathrm{pre}_R(X)$, by Theorem 5.1 we have that $\mathbf{EF}^A = \lambda S^\sharp.\mathrm{lfp}(\lambda X^\sharp.f^A(S^\sharp, X^\sharp)) : \wp(P) \to \wp(P)$. Moreover, as discussed above, $f^A(S^\sharp, X^\sharp) = \alpha_P(f(\gamma_P(S^\sharp), \gamma_P(X^\sharp))) = S^\sharp \cup_P \mathrm{pre}_{R^{\exists\exists}}(X^\sharp)$ so that $\mathbf{EF}^A = \lambda S^\sharp.\, \mathrm{lfp}(\lambda X^\sharp.S^\sharp \cup \mathrm{pre}_{R^{\exists\exists}}(X^\sharp))$, namely the best correct approximation $\mathbf{EF}^A$ can be computed as the least fixpoint characterization of the "finally" operator on the above abstract Kripke structure $\mathcal{A}$. □

6 Applications

We are mainly interested in applying Theorem 5.1 to standard fixpoint-based operators of well known temporal languages (cf. [3]), as recalled in Table 1.

Table 1. Temporal operators in fixpoint form

"Finally"	$\mathbf{AF}(S) = \mathrm{lfp}(\lambda X.S \cup \mathbf{AX}(X))$
	$\mathbf{EF}(S) = \mathrm{lfp}(\lambda X.S \cup \mathbf{EX}(X))$
"Globally"	$\mathbf{AG}(S) = \mathrm{gfp}(\lambda X.S \cap \mathbf{AX}(X))$
	$\mathbf{EG}(S) = \mathrm{gfp}(\lambda X.S \cap \mathbf{EX}(X))$
"(Strong) Until"	$\mathbf{AU}(S, T) = \mathrm{lfp}(\lambda X.T \cup (S \cap \mathbf{AX}(X)))$
	$\mathbf{EU}(S, T) = \mathrm{lfp}(\lambda X.T \cup (S \cap \mathbf{EX}(X)))$
"Weak Until"	$\mathbf{AU}_{\boldsymbol{w}}(S, T) = \mathrm{gfp}(\lambda X.T \cup (S \cap \mathbf{AX}(X)))$
	$\mathbf{EU}_{\boldsymbol{w}}(S, T) = \mathrm{gfp}(\lambda X.T \cup (S \cap \mathbf{EX}(X)))$
"(Weak) Release"	$\mathbf{AR}(S, T) = \mathrm{gfp}(\lambda X.T \cap (S \cup \mathbf{AX}(X)))$
	$\mathbf{ER}(S, T) = \mathrm{gfp}(\lambda X.T \cap (S \cup \mathbf{EX}(X)))$
"Strong Release"	$\mathbf{AR}_{\boldsymbol{s}}(S, T) = \mathrm{lfp}(\lambda X.T \cap (S \cup \mathbf{AX}(X)))$
	$\mathbf{ER}_{\boldsymbol{s}}(S, T) = \mathrm{lfp}(\lambda X.T \cap (S \cup \mathbf{EX}(X)))$

6.1 Partitioning Abstractions

Let $P \in \mathrm{Part}(\Sigma)$ be any partition and let $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha_P, \gamma_P, \wp(P)_\subseteq)$ be the corresponding partitioning GI. By Proposition 2.1 (i), $\mathcal{G}^\triangledown = (\wp(\Sigma)_\supseteq, \alpha_P^\triangledown, \gamma_P, \wp(P)_\supseteq)$ is a GI where $\alpha_P^\triangledown(S) = \{B \in P \mid B \subseteq S\}$. Hence, while $\mathcal{G}$ over-approximates a set S by the set of blocks in P which have a nonempty intersection with S, $\mathcal{G}^\triangledown$ under-approximates S by the set of blocks in P which are contained in S. Thus, we can apply Theorem 5.1 to $\mathcal{G}$ for least fixpoints and to $\mathcal{G}^\triangledown$ for greatest fixpoints. Since $\mathcal{G}$ is disjunctive, Let us note that by Proposition 2.1 (ii), $\mathcal{G}$ is forward complete for some function F iff $\mathcal{G}^\triangledown$ is forward complete for F. Hence, the hypotheses of Theorem 5.1 for least and greatest fixpoints actually coincide. Furthermore, in this case, the best correct approximations of F w.r.t. $\mathcal{G}$ and $\mathcal{G}^\triangledown$ coincide. In order to distinguish which GI has been applied, we use f^A to denote the best correct approximation of some concrete function f w.r.t. $\mathcal{G}$ while $f^{\triangledown A}$ denotes the best correct approximation of f w.r.t. $\mathcal{G}^\triangledown$.

For the standard temporal fixpoint-based operators in Table 1, the following result shows that their best correct approximations on a s.p. partitioning abstract domain preserve their characterizations as least/greatest fixpoints.

Corollary 6.1. *Let $P \in \mathrm{Part}(\Sigma)$ and $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha_P, \gamma_P, A = \wp(P)_\subseteq)$ be the corresponding partitioning GI. Assume that $\mathcal{G}$ is forward complete for some fixpoint-based operator F in Table 1. Then, the corresponding best correct approximations of F w.r.t. $\mathcal{G}$ are as follows:*

$$\begin{array}{l}
\mathbf{AF}^A(S^\sharp) = \mathrm{lfp}(\lambda X^\sharp . S^\sharp \cup_P \widetilde{\mathrm{pre}}_R^A(X^\sharp)) \\
\mathbf{EF}^A(S^\sharp) = \mathrm{lfp}(\lambda X^\sharp . S^\sharp \cup_P \mathrm{pre}_R^A(X^\sharp)) \\
\mathbf{AG}^A(S^\sharp) = \mathrm{gfp}(\lambda X^\sharp . S^\sharp \cap_P \widetilde{\mathrm{pre}}_R^{\triangledown A}(X^\sharp)) \\
\mathbf{EG}^A(S^\sharp) = \mathrm{gfp}(\lambda X^\sharp . S^\sharp \cap_P \mathrm{pre}_R^{\triangledown A}(X^\sharp)) \\
\mathbf{AU}^A(S^\sharp, T^\sharp) = \mathrm{lfp}(\lambda X^\sharp . T^\sharp \cup_P (S^\sharp \cap_P \widetilde{\mathrm{pre}}_R^A(X^\sharp))) \\
\mathbf{EU}^A(S^\sharp, T^\sharp) = \mathrm{lfp}(\lambda X^\sharp . T^\sharp \cup_P (S^\sharp \cap_P \mathrm{pre}_R^A(X^\sharp))) \\
\mathbf{AU}_{\boldsymbol{w}}^A(S^\sharp, T^\sharp) = \mathrm{gfp}(\lambda X^\sharp . T^\sharp \cup_P (S^\sharp \cap_P \widetilde{\mathrm{pre}}_R^{\triangledown A}(X^\sharp))) \\
\mathbf{EU}_{\boldsymbol{w}}^A(S^\sharp, T^\sharp) = \mathrm{gfp}(\lambda X^\sharp . T^\sharp \cup_P (S^\sharp \cap_P \mathrm{pre}_R^{\triangledown A}(X^\sharp))) \\
\mathbf{AR}^A(S^\sharp, T^\sharp) = \mathrm{gfp}(\lambda X^\sharp . T^\sharp \cap_P (S^\sharp \cup_P \widetilde{\mathrm{pre}}_R^{\triangledown A}(X^\sharp))) \\
\mathbf{ER}^A(S^\sharp, T^\sharp) = \mathrm{gfp}(\lambda X^\sharp . T^\sharp \cap_P (S^\sharp \cup_P \mathrm{pre}_R^{\triangledown A}(X^\sharp))) \\
\mathbf{AR}_{\boldsymbol{s}}^A(S^\sharp, T^\sharp) = \mathrm{lfp}(\lambda X^\sharp . T^\sharp \cap_P (S^\sharp \cup_P \widetilde{\mathrm{pre}}_R^A(X^\sharp))) \\
\mathbf{ER}_{\boldsymbol{s}}^A(S^\sharp, T^\sharp) = \mathrm{lfp}(\lambda X^\sharp . T^\sharp \cap_P (S^\sharp \cup_P \mathrm{pre}_R^A(X^\sharp)))
\end{array}$$

It turns out that the best correct approximations pre_R^A and $\widetilde{\mathrm{pre}}_R^{\triangledown A}$ can be characterized through the abstract transition relation $R^{\exists\exists} \subseteq P \times P$ as follows.

Lemma 6.2. $\mathrm{pre}_R^A = \mathrm{pre}_{R^{\exists\exists}}$ *and* $\widetilde{\mathrm{pre}}_R^{\triangledown A} = \widetilde{\mathrm{pre}}_{R^{\exists\exists}}$.

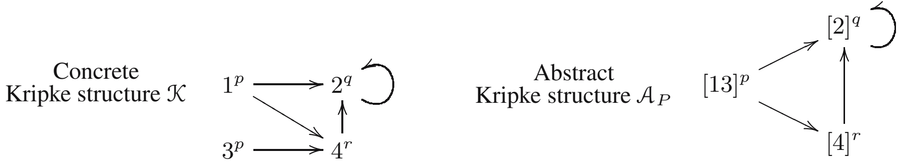
Let $Op \subseteq \{\mathrm{EX}, \mathrm{AX}, \mathrm{EF}, \mathrm{AG}, \mathrm{EU}, \mathrm{AU}_w, \mathrm{AR}, \mathrm{ER}_s\}$ be a set of temporal fixpoint-based operators and let $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid f(\varphi_1, \ldots, \varphi_{\mathrm{ar}(f)})$, where $f \in Op$, be the corresponding language. Let $\mathcal{K} = (\Sigma, R, AP, \ell)$ be a concrete Kripke structure and $\mathcal{S}$ be the concrete semantic structure for $\mathcal{L}$ induced by $\mathcal{K}$. Consider now a partition $P \in \mathrm{Part}(\Sigma)$ and the corresponding abstract semantic structure $\mathcal{S}^P = (\wp(P), I^P)$.

Assume that $\mathcal{S}^P$ is s.p. for $\mathcal{L}$. As a consequence of the above results, it turns out that one can define an abstract Kripke structure on P whose abstract transition relation is $R^{\exists\exists}$ which induces precisely $\mathcal{S}^P$.

Corollary 6.3. *If $\mathcal{S}^P$ is s.p. for $\mathcal{L}$ then $\mathcal{S}^P$ is induced by the abstract Kripke structure $\mathcal{A}_P = (P, R^{\exists\exists}, AP, \ell_P)$, where $\ell_P \stackrel{\text{def}}{=} \lambda B \in P.\{p \in AP \mid B \in I^P(p)\}$.*

Thus, a strongly preserving abstract model checking of the language $\mathcal{L}$ can be performed on the abstract Kripke structure $\mathcal{A}_P$.

Example 6.4. Let us consider $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{AG}\varphi$ and the following Kripke structure $\mathcal{K}$ and let $\mathcal{S}$ be the concrete semantic structure for $\mathcal{L}$ induced by $\mathcal{K}$.

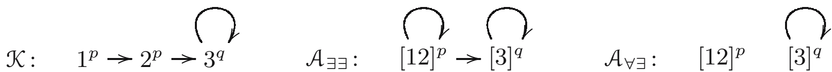


Let us consider the partition $P = \{[13], [2], [4]\}$ and the corresponding abstract semantic structure $\mathcal{S}^P = (\wp(P), I^P)$. It turns out that $\mathcal{S}^P$ is s.p. for $\mathcal{L}$. This is a consequence of the fact that the abstract domain $\wp(P)$ is operator-wise forward complete for $\mathcal{L}$ hence $\wp(P)$ is language forward complete for $\mathcal{L}$ and in turn, by Theorem 3.4, $\mathcal{S}^P$ is s.p. for $\mathcal{L}$. In fact, the following equalities show that $\wp(P)$ is forward complete for $\mathbf{AG}$, because $\mathbf{AG}$ maps unions of blocks in P to unions of blocks in P:

$$\mathbf{AG}(\varnothing) = \mathbf{AG}(\{4\}) = \mathbf{AG}(\{1,3\}) = \mathbf{AG}(\{1,3,4\}) = \varnothing;$$
$$\mathbf{AG}(\{2\}) = \mathbf{AG}(\{1,2,3\}) = \{2\};$$
$$\mathbf{AG}(\{2,4\}) = \{2,4\};$$
$$\mathbf{AG}(\{1,2,3,4\}) = \{1,2,3,4\}.$$

Thus, by Corollary 6.3, it turns out that $\mathcal{S}^P$ is induced by the abstract Kripke structure $\mathcal{A}_P = (P, R^{\exists\exists}, AP_P, \ell_P)$ which is depicted above. Let us notice that P is not a bisimulation on $\mathcal{K}$ because the states 1 and 3 belong to the same block $[13]$ and $1 \rightarrow 2$ while $3 \not\rightarrow 2$. Thus, strong preservation of $\mathcal{L}$ on the abstract Kripke structure $\mathcal{A}_P$, with abstract transition relation $R^{\exists\exists}$, cannot be obtained as a consequence of standard strong preservation results [2, 3, 13]. □

Example 6.5. Let us consider $\mathcal{L} \ni \varphi ::= p \mid \varphi \wedge \varphi_2 \mid \neg\varphi \mid \mathrm{EG}\varphi$ and the following Kripke structure $\mathcal{K}$ and let $\mathcal{S}$ be the concrete semantic structure for $\mathcal{L}$ induced by $\mathcal{K}$.



In this case, EG is not included among the operators of Corollary 6.3. Let us consider the partition $P = \{[13], [2], [4]\}$, the abstract domain $A = \wp(P)$ and the corresponding abstract semantic structure $\mathcal{S}^A = (A, I^A)$. It turns out that $\mathcal{S}^A$ is s.p. for $\mathcal{L}$. As in Example 6.4, by Theorem 3.4, this derives from the following equalities which show

that A is forward complete for $\mathbf{EG}$, because $\mathbf{EG}$ maps unions of blocks in P to unions of blocks in P:

$$\mathbf{EG}(\varnothing) = \mathbf{EG}(\{1,2\}) = \varnothing; \quad \mathbf{EG}(\{3\}) = \{3\}; \quad \mathbf{EG}(\{1,2,3\}) = \{1,2,3\}.$$

Let us point out here that both the abstract Kripke structures $\mathcal{A}_{\exists\exists}$ and $\mathcal{A}_{\forall\exists}$ on P depicted above, whose abstract transition relations are, respectively, $R^{\exists\exists}$ and $R^{\forall\exists}$, are not s.p. for $\mathcal{L}$. This is shown by the following counterexamples:

$$[1,2] \models^{\mathcal{A}_{\exists\exists}} \mathrm{EG}p \text{ while } 1 \not\models^{\mathcal{K}} \mathrm{EG}p; \quad [1,2] \not\models^{\mathcal{A}_{\forall\exists}} \mathrm{EG}(p \vee q) \text{ while } 1 \models^{\mathcal{K}} \mathrm{EG}(p \vee q).$$

On the other hand, we can exploit Corollary 6.1 so that $\mathbf{EG}^A(S^\sharp) = \mathrm{gfp}(\lambda X^\sharp . S^\sharp \cap_P \mathrm{pre}_R^{\triangledown A}(X^\sharp))$, where $\mathrm{pre}_R^{\triangledown A} = \alpha_P^{\triangledown} \circ \mathrm{pre}_R \circ \gamma_P$. For instance, we have that

$$\mathrm{pre}_R^{\triangledown A}(\{[3]\}) = \alpha_P^{\triangledown}(\mathrm{pre}_R(\gamma_P(\{[3]\}))) = \alpha_P^{\triangledown}(\mathrm{pre}_R(\{3\})) = \alpha_P^{\triangledown}(\{2,3\}) = \{[3]\}.$$

Likewise, $\mathrm{pre}_R^{\triangledown A}(\varnothing) = \mathrm{pre}_R^{\triangledown A}(\{[12]\}) = \varnothing$ and $\mathrm{pre}_R^{\triangledown A}(\{[12],[3]\}) = \{[12],[3]\}$. As an example, we have that $\mathbf{EG}^A(\{[3]\}) = \mathrm{gfp}(\lambda X^\sharp . \{[3]\} \cap \mathrm{pre}_R^{\triangledown A}(X^\sharp)) = \{[3]\}$. $\square$

6.2 Disjunctive Abstractions

In model checking, disjunctive abstract domains have been implicitely used by Henzinger et al.'s [14] algorithm for computing simulation equivalence: in fact, this algorithm maintains, for any state $s \in \Sigma$, a set of states $\mathrm{sim}(s) \subseteq \Sigma$ which represents exactly a disjunctive abstract domain. As observed in Section 3.2, any partitioning abstract domain is disjunctive while the converse is not true.

Let $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ be a disjunctive GI. By Proposition 2.1 (i), $\mathcal{G}^\triangledown = (\wp(\Sigma)_\supseteq, \alpha^\triangledown, \gamma, A_\geq)$ is a GI where $\alpha^\triangledown(S) = \vee\{a \in A \mid \gamma(a) \subseteq S\}$. Thus, we can apply Theorem 5.1 to $\mathcal{G}$ for least fixpoints and Theorem 5.1 to $\mathcal{G}^\triangledown$ for greatest fixpoints. Also, as already observed in Section 6.1, the hypotheses of Theorem 5.1 for least and greatest fixpoints coincide and, in this case, the best correct approximations of some concrete function w.r.t. $\mathcal{G}$ and $\mathcal{G}^\triangledown$ coincide. We use f^A to denote the best correct approximation of some concrete function f w.r.t. $\mathcal{G}$ while $f^{\triangledown A}$ denotes the best correct approximation of f w.r.t. $\mathcal{G}^\triangledown$. Here, we can generalize Corollary 6.1 to disjunctive abstract domains for the case of "finally" and "globally" operators.

Corollary 6.6. *Let $\mathcal{G} = (\wp(\Sigma)_\subseteq, \alpha, \gamma, A)$ be a disjunctive GI .Assume that $\mathcal{G}$ is forward complete for some operator $F \in \{\mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}\}$. Then, the corresponding best correct approximations of F w.r.t. $\mathcal{G}$ are as follows:*

$$\mathbf{AF}^A(S^\sharp) = \mathrm{lfp}(\lambda X^\sharp . S^\sharp \cup \widetilde{\mathrm{pre}}_R^A(X^\sharp)); \qquad \mathbf{EF}^A(S^\sharp) = \mathrm{lfp}(\lambda X^\sharp . S^\sharp \cup \mathrm{pre}_R^A(X^\sharp));$$
$$\mathbf{AG}^A(S^\sharp) = \mathrm{gfp}(\lambda X^\sharp . S^\sharp \cap \widetilde{\mathrm{pre}}_R^{\triangledown A}(X^\sharp)); \quad \mathbf{EG}^A(S^\sharp) = \mathrm{gfp}(\lambda X^\sharp . S^\sharp \cap \mathrm{pre}_R^{\triangledown A}(X^\sharp)).$$

Example 6.7. Let us consider the concrete Kripke structure $\mathcal{K}$ of Example 5.2 and the language $\mathcal{L} \ni \varphi ::= p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathrm{EF}\varphi$. Let $\mathrm{Atoms} \stackrel{\mathrm{def}}{=} \{[1],[2],[3],[6],[245]\}$ and let A be the closure under arbitrary unions of Atoms. Let $(\wp(\Sigma)_\subseteq, \alpha, \mathrm{id}, A_\subseteq)$ be the corresponding disjunctive GI where α on singletons in $\wp(\Sigma)$ is as follows:

$$\alpha(\{1\}) = [1]; \qquad \alpha(\{2\}) = [2]; \qquad \alpha(\{3\}) = [3];$$
$$\alpha(\{4\}) = [245]; \qquad \alpha(\{5\}) = [245]; \qquad \alpha(\{6\}) = [6].$$

It turns out that A is forward complete for **EF** because **EF** maps atoms to unions of atoms and **EF** is additive:

$$\mathbf{EF}(\{1\}) = \{1\};\ \ \mathbf{EF}(\{2\}) = \{1,2\};\ \ \mathbf{EF}(\{3\}) = \{1,2,3\};$$
$$\mathbf{EF}(\{6\}) = \{1,2,3,4,5,6\};\ \ \mathbf{EF}(\{2,4,5\}) = \{1,2,3,4,5\}.$$

Thus, we can apply Corollary 6.6 so that $\mathbf{EF}^A(S^\sharp) = \mathrm{lfp}(\lambda X^\sharp.S^\sharp \cup \mathrm{pre}_R^A(X^\sharp))$, where $\mathrm{pre}_R^A = \alpha \circ \mathrm{pre}_R \circ \mathrm{id}$. For instance, pre_R^A on the atom [245] is as follows:

$$\mathrm{pre}_R^A([245]) = \alpha(\mathrm{pre}_R(\{2,4,5\})) = \alpha(\{1,3,4\}) = [12345].$$

Likewise, we have that pre_R^A on Atoms is as follows:

$$\mathrm{pre}_R^A([1]) = \varnothing;\ \ \mathrm{pre}_R^A([2]) = [1];\ \ \mathrm{pre}_R^A([3]) = [2];\ \ \mathrm{pre}_R^A([6]) = [2456].$$

As an example, $\mathbf{EF}^A([6]) = \mathrm{lfp}(\lambda X^\sharp.[6] \cup \mathrm{pre}_R^A(X^\sharp))$ is computed as follows:

$$\begin{aligned}
X_0^\sharp &= \varnothing;\\
X_1^\sharp &= [6] \cup \mathrm{pre}_R^A(\varnothing) = [6] \cup \varnothing = [6];\\
X_2^\sharp &= [6] \cup \mathrm{pre}_R^A([6]) = [6] \cup [2456] = [2456];\\
X_3^\sharp &= [6] \cup \mathrm{pre}_R^A([2456]) = [6] \cup [123456] = [123456] \quad \text{(fixpoint)}
\end{aligned}$$

How to obtain an abstract Kripke structure which is s.p. for $\mathcal{L}$? This can be obtained from the coarsest s.p. partition $P_\mathcal{L}$ for $\mathcal{L}$ (cf. Section 3.3). As a consequence of results in [17], it turns out that $P_\mathcal{L} = \{[1],[2],[3],[6],[45]\}$ because $\wp(P_\mathcal{L})$ is exactly the least partitioning refinement of A (cf. Section 3.2). One can define a s.p. abstract Kripke structure $\mathcal{A}$ on $P_\mathcal{L}$ by considering $R^{\exists\exists}$ as abstract transition relation:

$$[1]^p \longrightarrow [2]^q \longrightarrow [3]^p \longrightarrow [45]^q \longrightarrow [6]^r$$

For the abstract Kripke structure $\mathcal{A}$, $\mathbf{EF}^\sharp([6]) = \mathrm{lfp}(\lambda X^\sharp.\{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(X^\sharp))$ is computed as follows:

$$\begin{aligned}
X_0^\sharp &= \varnothing;\\
X_1^\sharp &= \{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(\varnothing) = \{[6]\};\\
X_2^\sharp &= \{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(\{[6]\}) = \{[6]\} \cup \{[6],[45]\} = \{[6],[45]\};\\
X_3^\sharp &= \{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(\{[6],[45]\}) = \{[6]\} \cup \{[6],[45],[3]\} = \{[6],[45],[3]\};\\
X_4^\sharp &= \{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(\{[6],[45],[3]\}) = \{[6]\} \cup \{[6],[45],[3],[2]\} = \{[6],[45],[3],[2]\};\\
X_5^\sharp &= \{[6]\} \cup \mathrm{pre}_{R^{\exists\exists}}(\{[6],[45],[3],[2]\}) = \{[6]\} \cup \{[6],[45],[3],[2],[1]\}\\
&= \{[6],[45],[3],[2],[1]\} \quad \text{(fixpoint)}
\end{aligned}$$

The point to observe here is that this standard approach needs a greater number of iterations than our abstract interpretation-based approach to reach the fixpoint. □

Acknowledgements. This work was partially supported by the FIRB Project "Abstract interpretation and model checking for the verification of embedded systems" and by the COFIN2004 Project "AIDA: Abstract Interpretation Design and Applications".

References

1. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back, 10 Years Ahead*. LNCS 2000, pp. 176-194, 2001.
2. E.M. Clarke, O. Grumberg and D. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512-1542, 1994.
3. E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. The MIT Press, 1999.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, 238-252, 1977.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th ACM POPL*, 269-282, 1979.
6. P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering Journal*, 6(1):69-95, 1999.
7. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. 27th ACM POPL*, pp. 12-25, 2000.
8. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD Thesis, Eindhoven Univ., 1996.
9. D. Dams. Flat fragments of CTL and CTL*: separating the expressive and distinguishing powers. *Logic J. of the IGPL*, 7(1):55-78, 1999.
10. D. Dams, O. Grumberg and R. Gerth. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 16(5):1512-1542, 1997.
11. R. Giacobazzi and E. Quintarelli. Incompleteness, counterexamples and refinements in abstract model checking. In *Proc. 8th SAS*, LNCS 2126, pp. 356-373, 2001.
12. R. Giacobazzi, F. Ranzato and F. Scozzari. Making abstract interpretations complete. *J. ACM*, 47(2):361-416, 2000.
13. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843-871, 1994.
14. M.R. Henzinger, T.A. Henzinger and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. 36th FOCS*, pp. 453-462, 1995.
15. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1-36, 1995.
16. D. Massé. Semantics for abstract interpretation-based static analyzes of temporal properties. In *Proc. 9th SAS*, LNCS 2477, pp. 428-443, 2002.
17. F. Ranzato and F. Tapparo. Strong preservation as completeness in abstract interpretation. In *Proc. 13th ESOP*, LNCS 2986, pp. 18-32, 2004.
18. F. Ranzato and F. Tapparo. An abstract interpretation-based refinement algorithm for strong preservation. In *Proc. 11th TACAS*, LNCS 3440, pp. 140-156, 2005.
19. D.A. Schmidt. Closed and logical relations for over- and under-approximation of powersets. In *Proc. 11th SAS*, LNCS 3148, pp. 22-37, 2004.

Symbolic Methods to Enhance the Precision of Numerical Abstract Domains*

Antoine Miné

École Normale Supérieure, Paris, France
mine@di.ens.fr
http://www.di.ens.fr/~mine

Abstract. We present lightweight and generic symbolic methods to improve the precision of numerical static analyses based on Abstract Interpretation. The main idea is to simplify numerical expressions before they are fed to abstract transfer functions. An important novelty is that these simplifications are performed on-the-fly, using information gathered dynamically by the analyzer.

A first method, called "linearization," allows abstracting arbitrary expressions into affine forms with interval coefficients while simplifying them. A second method, called "symbolic constant propagation," enhances the simplification feature of the linearization by propagating assigned expressions in a symbolic way. Combined together, these methods increase the relationality level of numerical abstract domains and make them more robust against program transformations. We show how they can be integrated within the classical interval, octagon and polyhedron domains. These methods have been incorporated within the Astrée static analyzer that checks for the absence of run-time errors in embedded critical avionics software. We present an experimental proof of their usefulness.

1 Introduction

Ensuring the correctness of software is a difficult but important task, especially in embedded critical applications such as planes or rockets. There is currently a great need for static analyzers able to provide invariants automatically and directly on the source code. As the strongest invariants are not computable in general, such tools need to perform sound approximations at the expense of completeness. In this article, we will only consider the properties of numerical variables and work in the Abstract Interpretation framework. A static analyzer is thus parameterized by a *numerical abstract domain*, that is, a set of computer-representable numerical properties together with algorithms to compute the semantics of program instructions.

There already exist quit a few numerical abstract domains. Well-known examples include the interval domain [5] that discovers variable bounds, and the polyhedron domain [8] for affine inequalities. Each domain achieves some cost

* This work was partially supported by the Astrée RNTL project and the APRON project from the ACI "Sécurité & Informatique."

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 348–363, 2006.
© Springer-Verlag Berlin Heidelberg 2006

$X \leftarrow [-10, 20];$
$Y \leftarrow X;$
if $(Y \leq 0)$ { $Y \leftarrow -X;$ }
// *here,* $Y \in [0, 20]$

Fig. 1. Absolute value computation example

$X \leftarrow [0, 1];$
$Y \leftarrow [0, 0.1];$
$Z \leftarrow [0, 0.2];$
$T \leftarrow (X \times Y) - (X \times Z) + Z;$
// *here,* $T \in [0, 0.2]$

Fig. 2. Linear interpolation computation example

versus precision balance. In particular, non-relational domains—*e.g.*, the interval domain—are much faster but also much less precise than relational domains—able to discover variable relationships. Although the interval information seem sufficient—it allows expressing most correctness requirements, such as the absence of arithmetic overflows or out-of-bound array accesses—relational invariants are often necessary during the course of the analysis to find tight bounds. Consider, for instance, the program of Fig. 1 that computes the absolute value of X. We expect the analyzer to infer that, at the end of the program, $Y \in [0, 20]$. The interval domain will find the coarser result $Y \in [-20, 20]$ because it cannot exploit the information $Y = X$ during the test $Y \leq 0$. The polyhedron domain is precise enough to infer the tightest bounds, but results in a loss of efficiency. In our second example, Fig. 2, T is linearly interpolated between Y and Z, thus, we have $T \in [0, 0.2]$. Using plain interval arithmetics, one finds the coarser result $T \in [-0.2, 0.3]$. As the assignment in T is not affine, the polyhedron domain cannot perform any better.

In this paper, we present symbolic enhancement techniques that can be applied to abstract domains to solve these problems and increase their robustness against program transformations. In Fig. 1, our *symbolic constant propagation* is able to propagate the information $Y = X$ and discover tight bounds using only the interval domain. In Fig. 2, our *linearization* technique allows us to prove that $T \in [0, 0.3]$ using the interval domain (this result is not optimal, but still much better than $T \in [-0.2, 0.3]$). The techniques are generic and can be applied to other domains, such as the polyhedron domain. However, the improvement varies greatly from one example to another and enhanced domains do not enjoy best abstraction functions. Thus, our techniques depend upon *strategies*, some of which are proposed in the article.

Related Work. Our linearization can be related to *affine arithmetics*, a technique introduced by Vinícius et al. in [16] to refine interval arithmetics by taking into account existing correlations between computed quantities. Both use a symbolic form with linear properties to allow basic algebraic simplifications. The main difference is that we relate directly program variables while affine arithmetics

introduces synthetic variables. This allows us to treat control flow joins and loops, and to interact with relational domains, which is not possible with affine arithmetics. Our linearization was first introduced in [13] to abstract floating-point arithmetics. It is presented here with some improvements—including the introduction of several strategies.

Our symbolic constant propagation technique is similar to the classical constraint propagation proposed by Kildall in [11] to perform optimization. However, scalar constants are replaced with expression trees, and our goal is not to improve the efficiency but the precision of the abstract execution. It is also related to the work of Colby: he introduces, in [4], a language of transfer relations to propagate, combine and simplify, in a fully symbolic way, sequences of transfer functions. We are more modest as we do not handle disjunctions symbolically and do not try to infer symbolic loop invariants. Instead, we rely on the underlying numerical abstract domain to perform most of the semantical job. A major difference is that, while Colby's framework statically transforms the abstract equation system to be solved by the analyzer, our framework performs this transformation on-the-fly and benefits from the information dynamically inferred by the analyzer.

Overview of the Paper. The paper is organised as follows. In Sect. 2, we introduce a language—much simplified for the sake of illustration—and recall how to perform a numerical static analysis parameterized by an abstract domain. Sect. 3 then explains how symbolic expression manipulations can be soundly incorporated within the analysis. Two symbolic methods are then introduced: expression linearization, in Sect. 4, and symbolic constant propagation, in Sect. 5. Sect. 6 discusses our practical implementation within the ASTRÉE static analyzer and presents some experimental results. We conclude in Sect. 7.

2 Framework

In this section, we briefly recall the classical design of a static analyzer using the Abstract Interpretation framework by Cousot and Cousot [6, 7]. This design is specialised towards the automatic computation of *numerical* invariants, and thus, is parameterized by a numerical abstract domain.

2.1 Syntax of the Language

For the sake of presentation, we will only consider in this article a very simplified programming language focusing on manipulating numerical variables. We suppose that a program manipulates only a fixed, finite set of n variables, $\mathcal{V} \stackrel{\text{def}}{=} \{V_1, \ldots, V_n\}$, with values within a perfect mathematical set, $\mathbb{I} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$. A program $P \in \mathcal{P}(\mathcal{L} \times \mathit{inst} \times \mathcal{L})$ is a single control-flow graph where nodes are program points, in $\mathcal{L}$, and arcs are labelled by instructions in *inst*. We denote by e the entry program point. As described in Fig. 3, only two types of instructions are allowed: assignments ($X \leftarrow \mathit{expr}$) and tests ($\mathit{expr} \bowtie 0$?), where *expr* are numerical expressions and $\bowtie$ is a comparison operator. In the syntax of expressions, classical numerical constants have been replaced with *intervals* $[a, b]$

$$
\begin{array}{llll}
\textit{expr} ::= & X & & X \in \mathcal{V} \\
& \mid & [a, b] & a \in \mathbb{I} \cup \{-\infty\},\ b \in \mathbb{I} \cup \{+\infty\},\ a \leq b \\
& \mid & \textit{expr} \diamond \textit{expr} & \diamond \in \{+, -, \times, /\} \\
\\
\textit{inst} ::= & X \leftarrow \textit{expr} & & X \in \mathcal{V} \\
& \mid & \textit{expr} \bowtie 0\ ? & \bowtie \in \{=, \neq, <, \leq, \geq, >\}
\end{array}
$$

Fig. 3. Syntax of our simple language

with constant bounds—possibly $+\infty$ or $-\infty$. Such intervals correspond to a non-deterministic choice of a new value within the bounds each time the expression is evaluated. This will be key in defining the concept of *expression abstraction* in Sects. 3–5. Moreover, interval constants appear naturally in programs that fetch input values from an external environment, or when modeling rounding errors in floating-point computations.

Affine forms play an important role in program analysis as they are easy to manipulate and appear frequently as program invariants. We enhance affine forms with the non-determinism of intervals by defining *interval affine forms* as the expressions of the form: $[a_0, b_0] + \sum_k ([a_k, b_k] \times V_k)$.

2.2 Concrete Semantics of the Language

The *concrete semantics* of a program is the most precise mathematical expression of its behavior. Let us first define an *environment* as a function, in $\mathcal{V} \to \mathbb{I}$, associating a value to each variable. We choose a simple *invariant semantics* that associates to each program point $l \in \mathcal{L}$ the set of all environments $\mathcal{X}_l \in \mathcal{P}(\mathcal{V} \to \mathbb{I})$ that can hold when l is reached. Given an environment $\rho \in (\mathcal{V} \to \mathbb{I})$, the semantics $[\![\, \textit{expr} \,]\!](\rho)$ of an expression *expr*, shown in Fig. 4, is the set of values the expression can evaluate to. It outputs a set to account for non-determinism. When $\mathbb{I} = \mathbb{Z}$, the *truncate* function rounds the possibly non-integer result of the division towards an integer by *truncation*, as it is common in most computer languages. Divisions by zero are undefined, that is, return no result; for the sake of simplicity, we have not introduced any error state. The semantics of assignments and tests is defined by *transfer functions* $\{\!|\, \textit{inst} \,|\!\} : \mathcal{P}(\mathcal{V} \to \mathbb{I}) \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$ in Fig. 4. The assignment transfer function returns environments where one variable has changed its value ($\rho[V \mapsto x]$ denotes the function equal to ρ on

$$
\begin{array}{lcll}
[\![\, X \,]\!](\rho) & \stackrel{\text{def}}{=} & \{\, \rho(X) \,\} & \\
[\![\, [a,b] \,]\!](\rho) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{I} \mid a \leq x \leq b \,\} & \\
[\![\, e_1 \diamond e_2 \,]\!](\rho) & \stackrel{\text{def}}{=} & \{\, x \diamond y \mid x \in [\![\, e_1 \,]\!](\rho),\ y \in [\![\, e_2 \,]\!](\rho) \,\} \quad \diamond \in \{+, -, \times\} & \\
[\![\, e_1 / e_2 \,]\!](\rho) & \stackrel{\text{def}}{=} & \{\, \textit{truncate}(x/y) \mid x \in [\![\, e_1 \,]\!](\rho),\ y \in [\![\, e_2 \,]\!](\rho),\ y \neq 0 \,\} & \text{if } \mathbb{I} = \mathbb{Z} \\
[\![\, e_1 / e_2 \,]\!](\rho) & \stackrel{\text{def}}{=} & \{\, x/y \mid x \in [\![\, e_1 \,]\!](\rho),\ y \in [\![\, e_2 \,]\!](\rho),\ y \neq 0 \,\} & \text{if } \mathbb{I} \neq \mathbb{Z} \\
\\
\{\!|\, X \leftarrow e \,|\!\}(R) & \stackrel{\text{def}}{=} & \{\, \rho[X \mapsto v] \mid \rho \in R,\ v \in [\![\, e \,]\!](\rho) \,\} & \\
\{\!|\, e \bowtie 0\ ? \,|\!\}(R) & \stackrel{\text{def}}{=} & \{\, \rho \mid \rho \in R \text{ and } \exists v \in [\![\, e \,]\!](\rho),\ v \bowtie 0 \text{ holds} \,\} &
\end{array}
$$

Fig. 4. Concrete semantics

$\mathcal{V} \setminus \{V\}$ and that maps V to x). The test transfer function filters environments to keep only those that *may* satisfy the test. We can now define the semantics $(\mathcal{X}_l)_{l \in \mathcal{L}}$ of a program P as the smallest solution of the following equation system:

$$\begin{cases} \mathcal{X}_e = & V \to \mathbb{I} \\ \mathcal{X}_l = & \displaystyle\bigcup_{(l',i,l) \in P} \{\!| \, i \, |\!\}(\mathcal{X}_{l'}) \quad \text{when } l \neq e \end{cases} \tag{1}$$

It describes the *strongest invariant* at each program point.

2.3 Abstract Interpretation and Numerical Abstract Domains

The concrete semantics is very precise but cannot be computed fully automatically by a computer. We will only try to compute a sound overapproximation, that is, a *superset* of the environments reached by the program. We use Abstract Interpretation [6, 7] to design such an approximation.

Numerical Abstract Domains. An analysis is parameterized by a numerical abstract domain that allows representing and manipulating selected subsets of environments. Formally it is defined as:

– a set of computer-representable *abstract* elements $\mathcal{D}^\sharp$,
– a *partial order* $\sqsubseteq^\sharp$ on $\mathcal{D}^\sharp$ to model the relative precision of abstract elements,
– a monotonic *concretization* $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{V} \to \mathbb{I})$, that assigns a concrete property to each abstract element,
– a greatest element $\top^\sharp$ for $\sqsubseteq^\sharp$ such that $\gamma(\top^\sharp) = (\mathcal{V} \to \mathbb{I})$,
– *sound* and computable abstract versions $\{\!| \, inst \, |\!\}^\sharp$ of all transfer functions,
– *sound* and computable abstractions $\cup^\sharp$ and $\cap^\sharp$ of $\cup$ and $\cap$,
– a widening operator $\triangledown^\sharp$ if $\mathcal{D}^\sharp$ has infinite increasing chains.

The *soundness condition* for the abstraction $F^\sharp : (\mathcal{D}^\sharp)^n \to \mathcal{D}^\sharp$ of a $n-$ary operator F is: $F(\gamma(X_1^\sharp), \ldots, \gamma(X_n^\sharp)) \subseteq \gamma(F^\sharp(X_1^\sharp, \ldots, X_n^\sharp))$. It ensures that $F^\sharp$ does not forget any of F's behaviors. It can, however, introduce spurious ones.

Abstract Analysis. Given an abstract domain, an abstract version ($1^\sharp$) of the equation system (1) can be derived as:

$$\begin{cases} \mathcal{X}_e^\sharp = & \top^\sharp \\ \mathcal{X}_l^\sharp \sqsubseteq^\sharp & \displaystyle\bigcup_{(l',i,l) \in P}^{\sharp} \{\!| \, i \, |\!\}^\sharp(\mathcal{X}_{l'}^\sharp) \quad \text{when } l \neq e \end{cases} \tag{$1^\sharp$}$$

The soundness condition ensures that any solution of ($1^\sharp$) satisfies $\forall l \in \mathcal{L},\ \gamma(\mathcal{X}_l^\sharp) \supseteq \mathcal{X}_l$. The system can be solved by iterations, using a widening operator $\triangledown^\sharp$ to ensure termination. We refer the reader to Bourdoncle [2] for an in-depth description of possible iteration strategies. The computed $\mathcal{X}_l^\sharp$ is almost never the best abstraction—if it exists—of the concrete solution $\mathcal{X}_l$. Unavoidable losses of precision come from the use of convergence acceleration $\triangledown^\sharp$, non-necessarily best abstract transfer functions, and the fact that the composition of

best abstractions is generally not a best abstraction. This last issue explains why even the simplest semantics-preserving program transformations can drastically affect the quality of a static analysis.

Existing Numerical Domains. There exists many numerical abstract domains. We will be mostly interested in those able to express variable bounds. Such abstract domains include the well-known interval domain [5] (able to express invariants of the form $\bigwedge_i V_i \in [a_i, b_i]$), and the polyhedron domain [8] (able to express affine inequalities $\bigwedge_i \sum_j \alpha_{ij} V_i \geq \beta_j$). More recent domains, in-between these two in terms of cost and precision, include the octagon domain [12] ($\bigwedge_{ij} \pm V_i \pm V_j \leq c_{ij}$), the octahedron domain [3] ($\bigwedge_j \sum_i \alpha_{ij} V_i \geq \beta_j$ where $\alpha_{ij} \in \{-1, 0, 1\}$), and the Two Variable Per Inequality domain [15] ($\bigwedge_i \alpha_i V_{k_i} + \beta_i V_{l_i} \leq c_i$).

3 Incorporating Symbolic Methods

We suppose that we are given a numerical abstract domain $\mathcal{D}^\sharp$. The gist of our method is to replace, in the abstract transfer functions $\{\!|\, X \leftarrow e \,|\!\}^\sharp$ and $\{\!|\, e \bowtie 0\ ? \,|\!\}^\sharp$, each expression e with another one e', in a sound way.

Partial Order on Expressions. To define formally the notion of sound expression abstraction, we first introduce an approximation order $\preceq$ on expressions. A natural choice is to consider the point-wise ordering of the concrete semantics $[\![\cdot]\!]$ defined in Fig. 4, that is: $e_1 \preceq e_2 \overset{\text{def}}{\iff} \forall \rho \in (\mathcal{V} \to \mathbb{I}),\ [\![e_1]\!](\rho) \subseteq [\![e_2]\!](\rho)$. However, requiring the inclusion to hold for *all* environments is quite restrictive. More aggressive expression transformations can be enabled by only requiring soundness with respect to selected sets of environments. Our partial order $\preceq$ is now defined "up to" a set of environments $R \in \mathcal{P}(\mathcal{V} \to \mathbb{I})$:

Definition 1. $R \models e_1 \preceq e_2 \overset{\text{def}}{\iff} \forall \rho \in R,\ [\![e_1]\!](\rho) \subseteq [\![e_2]\!](\rho)$.

We denote by $R \models e_1 = e_2$ the associated equality relation.

Sound Symbolic Transformations. We wish now to abstract some transfer function, *e.g.*, $\{\!|\, V \leftarrow e \,|\!\}$, on an abstract environment $R^\sharp \in \mathcal{D}^\sharp$. The following theorem states that, if e' overapproximates e on $\gamma(R^\sharp)$, it is sound to replace e with e' in the abstract transfer functions:

Theorem 1. *If* $\gamma(R^\sharp) \models e \preceq e'$*, then:*
- $(\{\!|\, V \leftarrow e \,|\!\} \circ \gamma)(R^\sharp) \subseteq (\gamma \circ \{\!|\, V \leftarrow e' \,|\!\}^\sharp)(R^\sharp)$,
- $(\{\!|\, e \bowtie 0\ ? \,|\!\} \circ \gamma)(R^\sharp) \subseteq (\gamma \circ \{\!|\, e' \bowtie 0\ ? \,|\!\}^\sharp)(R^\sharp)$.

4 Expression Linearization

Our first symbolic transformation is an abstraction of arbitrary expressions into interval affine forms $i_0 + \sum_k (i_k \times V_k)$, where the i's stand for intervals.

4.1 Definitions

Interval Affine Form Operators. We first introduce a few operators to manipulate interval affine forms in a symbolic way. Using the classical interval arithmetic operators—denoted with a $\mathcal{I}$ superscript—we can define point-wisely the addition $\boxplus$ and subtraction $\boxminus$ of affine forms, as well as the multiplication $\boxtimes$ and division $\boxslash$ of an affine form by a constant interval:

Definition 2.
- $(i_0 + \sum_k i_k \times V_k) \boxplus (i'_0 + \sum_k i'_k \times V_k) \stackrel{\text{def}}{=} (i_0 +^{\mathcal{I}} i'_0) + \sum_k (i_k +^{\mathcal{I}} i'_k) \times V_k$,
- $(i_0 + \sum_k i_k \times V_k) \boxminus (i'_0 + \sum_k i'_k \times V_k) \stackrel{\text{def}}{=} (i_0 -^{\mathcal{I}} i'_0) + \sum_k (i_k -^{\mathcal{I}} i'_k) \times V_k$,
- $i \boxtimes (i_0 + \sum_k i_k \times V_k) \stackrel{\text{def}}{=} (i \times^{\mathcal{I}} i_0) + \sum_k (i \times^{\mathcal{I}} i_k) \times V_k$,
- $(i_0 + \sum_k i_k \times V_k) \boxslash i \stackrel{\text{def}}{=} (i_0 /^{\mathcal{I}} i) + \sum_k (i_k /^{\mathcal{I}} i) \times V_k$.

where the interval arithmetic operators are defined classically as:
- $[a,b] +^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [a+a', b+b']$,
- $[a,b] -^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [a-b', b-a']$,
- $[a,b] \times^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=} [\min(aa',ab',ba',bb'), \max(aa',ab',ba',bb')]$,
- $[a,b] /^{\mathcal{I}} [a',b'] \stackrel{\text{def}}{=}$
$$\begin{cases} [-\infty,+\infty] & \text{if } 0 \in [a',b'] \\ [\min(a/a',a/b',b/a',b/b'),\ \max(a/a',a/b',b/a',b/b')] & \text{when } \mathbb{I} \neq \mathbb{Z} \\ [\lfloor \min(a/a',a/b',b/a',b/b') \rfloor,\ \lceil \max(a/a',a/b',b/a',b/b') \rceil] & \text{when } \mathbb{I} = \mathbb{Z} \end{cases}$$

The following theorem states that these operators are always sound and, in some cases, complete—*i.e.*, $\preceq$ can be replaced by $=$:

Theorem 2. *For all interval affine forms l_1, l_2 and interval i, we have:*
- $\mathbb{I}^{\mathcal{V}} \models l_1 + l_2 = l_1 \boxplus l_2$,
- $\mathbb{I}^{\mathcal{V}} \models l_1 - l_2 = l_1 \boxminus l_2$,
- $\mathbb{I}^{\mathcal{V}} \models i \times l_1 = i \boxtimes l_1$, *if* $\mathbb{I} \neq \mathbb{Z}$,
- $\mathbb{I}^{\mathcal{V}} \models i \times l_1 \preceq i \boxtimes l_1$, *otherwise,*
- $\mathbb{I}^{\mathcal{V}} \models l_1 / i = l_1 \boxslash i$, *if* $\mathbb{I} \neq \mathbb{Z}$ *and* $0 \notin i$,
- $\mathbb{I}^{\mathcal{V}} \models l_1 / i \preceq l_1 \boxslash i$, *otherwise.*

When $\mathbb{I} = \mathbb{Z}$, we must conservatively round upper and lower bounds respectively towards $+\infty$ and $-\infty$ to ensure that Thm. 2 holds. The non-exactness of the multiplication and division can then lead to some precision degradation. For instance, $(X \boxslash 2) \boxtimes 2$ evaluates to $[0,2] \times X$ as, when computing $X \boxslash 2$, the non-integral value $1/2$ must be abstracted into the integral interval $[0,1]$. One solution is to perform all computations in $\mathbb{R}$, keeping in mind that, due to truncation, $l/[a,b]$ should be interpreted when $0 \notin [a,b]$ as $(l \boxslash [a,b]) \boxplus [-1+x, 1-x]$, where $x = 1/\min(|a|,|b|)$. We then obtain the more precise result $X + [-1,1]$.

We now introduce a so-called "intervalization" operator, ι, to abstracts interval affine forms into intervals. Given an abstract environment, it evaluates the affine form using interval arithmetics. Suppose that $\mathcal{D}^\sharp$ provides us with projection operators $\pi_k : \mathcal{D}^\sharp \to \mathcal{P}(\mathbb{I})$ able to return an interval overapproximation for each variable V_k. We define ι as:

Definition 3. $\iota(i_0 + \sum_k (i_k \times V_k))(R^\sharp) \stackrel{\text{def}}{=} i_0 +^{\mathcal{I}} \sum_k^{\mathcal{I}} (i_k \times^{\mathcal{I}} \pi_k(R^\sharp))$, *where each $\pi_k(R^\sharp)$ is an interval containing $\{\ \rho(V_k) \mid \rho \in \gamma(R^\sharp)\ \}$.*

The following theorem states that ι is a sound operator with respect to $R^\sharp$:

Theorem 3. *$\gamma(R^\sharp) \models l \preceq \iota(l)(R^\sharp)$.*

As π_k performs a non-relational abstraction, ι incurs a loss of precision whenever $\mathcal{D}^\sharp$ is a relational domain. Consider, for instance $R^\sharp$ such that $\gamma(R^\sharp) = \{ \rho \in (\{V_1, V_2\} \to [0,1]) \mid \rho(V_1) = \rho(V_2) \}$. Then, $[\![\iota(V_1 - V_2)(R^\sharp)]\!]$ is the constant function $[-1,1]$ while $[\![V_1 - V_2]\!]$ is 0.

Linearization. The linearization $(\!| e |\!)(R^\sharp)$ of an arbitrary expression e in an abstract environment $R^\sharp$ can now be defined by structural induction as follows:

Definition 4.
- $(\!| V |\!)(R^\sharp) \stackrel{\text{def}}{=} [1,1] \times V$,
- $(\!| [a,b] |\!)(R^\sharp) \stackrel{\text{def}}{=} [a,b]$,
- $(\!| e_1 + e_2 |\!)(R^\sharp) \stackrel{\text{def}}{=} (\!| e_1 |\!)(R^\sharp) \boxplus (\!| e_2 |\!)(R^\sharp)$,
- $(\!| e_1 - e_2 |\!)(R^\sharp) \stackrel{\text{def}}{=} (\!| e_1 |\!)(R^\sharp) \boxminus (\!| e_2 |\!)(R^\sharp)$,
- $(\!| e_1 / e_2 |\!)(R^\sharp) \stackrel{\text{def}}{=} (\!| e_1 |\!)(R^\sharp) \boxslash \iota((\!| e_2 |\!)(R^\sharp))(R^\sharp)$,
- $(\!| e_1 \times e_2 |\!)(R^\sharp) \stackrel{\text{def}}{=} \begin{cases} \textit{either} & \iota((\!| e_1 |\!)(R^\sharp))(R^\sharp) \boxtimes (\!| e_2 |\!)(R^\sharp) \\ \textit{or} & \iota((\!| e_2 |\!)(R^\sharp))(R^\sharp) \boxtimes (\!| e_1 |\!)(R^\sharp) \end{cases}$ *(see Sect. 4.3)*

The ι operator is used to deal with non-linear constructions: the right argument of a division and either argument of a multiplication are intervalized. As a consequence of Thms. 2 and 3, our linearization is sound:

Theorem 4. *$\gamma(R^\sharp) \models e \preceq (\!| e |\!)(R^\sharp)$.*

Obviously, $(\!| \cdot |\!)$ generally incurs a loss of precision with respect to $\preceq$. Also, $(\!| e |\!)$ is not monotonic in its e argument. Consider for instance X/X in the environment $R^\sharp$ such that $\pi_X(R^\sharp) = [1, +\infty]$. Although $\gamma(R^\sharp) \models X/X \preceq [1,1]$, we do not have $\gamma(R^\sharp) \models (\!| X/X |\!)(R^\sharp) \preceq (\!| [1,1] |\!)(R^\sharp)$ as $(\!| X/X |\!)(R^\sharp) = [0,1] \times X$. It is important to note that there is no useful notion of *best abstraction* of expressions for $\preceq$.

4.2 Integration with a Numerical Abstract Domain

Given an abstract domain, $\mathcal{D}^\sharp$, we can now derive a new abstract domain with linearization, $\mathcal{D}^\sharp_\mathcal{L}$, identical to $\mathcal{D}^\sharp$ except for the following transfer functions:

$$\begin{aligned} \{\!| V \leftarrow e |\!\}^\sharp_\mathcal{L}(R^\sharp) &\stackrel{\text{def}}{=} \{\!| V \leftarrow (\!| e |\!)(R^\sharp) |\!\}^\sharp(R^\sharp) \\ \{\!| e \bowtie 0\ ? |\!\}^\sharp_\mathcal{L}(R^\sharp) &\stackrel{\text{def}}{=} \{\!| (\!| e |\!)(R^\sharp) \bowtie 0\ ? |\!\}^\sharp(R^\sharp) \end{aligned}$$

The soundness of these transfer functions is guaranteed by Thms. 1 and 4.

Application to the Interval Domain. As all non-relational domains, the interval domain [5], is not able to exploit the fact that the same variable occurs several times in an expression. Our linearization performs some symbolic simplification, and so, is able to partly correct this problem. Consider, for instance, the assignment $\{\!| Y \leftarrow 3 \times X - X |\!\}$ in an abstract environment such that $X \in [a,b]$. The regular interval domain $\mathcal{D}^\mathcal{I}$ will assign $[3a-b, 3b-a]$ to Y, while $\mathcal{D}^\mathcal{I}_\mathcal{L}$ will assign $[2a, 2b]$ as $(\!| 3 \times X - X |\!)(R^\sharp) = 2 \times X$. This last answer is strictly more precise whenever $a \neq b$. Using the exactness of Thm. 2, one can prove that, when $\mathbb{I} \neq \mathbb{Z}$, the assignment in $\mathcal{D}^\mathcal{I}_\mathcal{L}$ is always more precise than in $\mathcal{D}^\mathcal{I}$. This may not be the case for a test, or when $\mathbb{I} = \mathbb{Z}$.

Application to the Octagon Domain. The octagon domain [12] is more precise than the interval one, but it is more complex. As a consequence, it is quite difficult to design abstract transfer functions for non-linear expressions. This problem can be solved by using our linearization in combination with the efficient and rather precise interval affine form abstract transfer functions proposed in our previous work [14]. The octagon domain with linearization is able to prove, for instance, that, after the assignment $X \leftarrow T \times Y$ in an environment such that $T \in [-1, 1]$, we have $-Y \leq X \leq Y$.

Application to the Polyhedron Domain. The polyhedron domain [8] is more precise than the octagon domain but cannot deal with full interval affine forms—only the constant coefficient may safely be an interval. To solve this problem, we introduce a function μ to abstract interval affine forms further by making all variable coefficients singletons. For the sake of conciseness, we give a formula valid only for $\mathbb{I} \neq Z$ and finite interval bounds:

Definition 5.
$$\mu\left([a_0, b_0] + \sum_k [a_k, b_k] \times V_k\right)(R^\sharp) \stackrel{\text{def}}{=}$$
$$\left([a_0, b_0] +^{\mathcal{I}} \sum_k^{\mathcal{I}} [(a_k - b_k)/2, (b_k - a_k)/2] \times^{\mathcal{I}} \pi_k(R^\sharp)\right) + \sum_k ((a_k + b_k)/2) \times V_k$$

μ works by "distributing" the weight $b_k - a_k$ of each variable coefficient into the constant component, using variable bounds information from $R^\sharp$. One can prove that μ is sound, that is, $\gamma(R^\sharp) \models l \preceq \mu(l)R^\sharp$.

Application to Floating-Point Arithmetics. Real-life programming languages do not manipulate rationals or reals, but floating-point numbers, which are much more difficult to abstract. Pervasive rounding must be taken into account. As most classical properties of arithmetic operators are no longer true, it is generally not safe to feed floating-point expressions to relational domains. One solution is to convert such expressions into real-valued expressions by making rounding explicit. Rounding is highly non-linear but can be abstracted using intervals. For instance, $X + Y$ in the floating-point world can be abstracted into $[1 - \epsilon_1, 1 + \epsilon_1] \times X + [1 - \epsilon_1, 1 + \epsilon_1] \times Y + [-\epsilon_2, \epsilon_2]$ using small constants ϵ_1 and ϵ_2 modeling, respectively, relative and absolute errors. This fits in our linearization framework which can be extended to treat soundly floating-point arithmetics. We refer the reader to related work [13] for more information.

4.3 Multiplication Strategies

When encountering a multiplication $e_1 \times e_2$ and neither $(\!| e_1 |\!)(R^\sharp)$ nor $(\!| e_2 |\!)(R^\sharp)$ evaluates to an interval, we must intervalize either argument. Both choices are valid, but influence greatly the precision of the result.

All-Cases Strategy. A first idea is to try both choices for each multiplication; we get a *set* of linearized expressions. We have no notion of greatest lower bound on expressions, so, we must evaluate a transfer function for all expressions in parallel, and take the intersection $\cap^\sharp$ of the resulting abstract elements in $\mathcal{D}^\sharp$.

Unfortunately, the cost is exponential in the number of multiplications in the original expression, hence the need for deterministic strategies that always select *one* interval affine form.

Interval-Size Strategy. A simple strategy is to intervalize the affine form that will yield the narrower interval. This greedy approach tries to limit the amplitude of the non-determinism introduced by multiplications. The extreme case holds when the amplitude of one interval is zero, meaning that the sub-expression is semantically a constant; intervalizing it will not result in any precision loss. Finally, note that the *relative* amplitude $(b-a)/|a+b|$ may be more significant than the absolute amplitude $b-a$ if we want to intervalize preferably expressions that are constant up to some small relative rounding error.

Simplification-Driven Strategy. Another idea is to maximize the amount of simplification by not intervalizing, when possible, sub-expressions containing variables appearing in other sub-expressions. For instance, in $X - (Y \times X)$, Y will be intervalized to yield $[1 - \max Y, 1 - \min Y] \times X$. Unlike the preceding greedy approach, this strategy is global and treats the expression as a whole.

Homogeneity Strategy. We now consider the linear interpolation of Fig. 2. In order to achieve the best precision, it is important to intervalize X in both multiplications. This yields $T \leftarrow [0,1] \times Y + [0,1] \times Z$ and we are able to prove that $T \geq 0$—however, we find that $T \leq 0.3$ while in fact $T \leq 0.2$. The interval-size strategy would choose to intervalize Y and Z that have smaller range than X, which yields the imprecise assignment $T \leftarrow [-0.2, 0.1] \times X + [0, 0.2]$. Likewise, the simplification-driven strategy may choose to keep X that appears in two sub-expressions and also intervalize both Y and Z. To solve this problem, we propose to intervalize the smallest set of variables that makes the expression homogeneous, that is, arguments of $+$ and $-$ operators should have the same degree. In order to make the $(1 - X)$ sub-expression homogeneous, X is intervalized. This last strategy is quite robust: it keeps working if we change the assignment into the equivalent $T \leftarrow X \times Y - X \times Z + Z$, or if we consider bi-linear interpolations or interpolations with normalization coefficients.

4.4 Concluding Remark

Our linearization is not equivalent to a static program transformation. To cope with non-linearity as best as we can, we exploit the information dynamically inferred by the analysis: first, in the intervalization ι, then, in the multiplication strategy. Both algorithms take as argument the current numerical abstract environment $R^\sharp$. As, dually, the linearization improves the precision of the next computed abstract element, the dynamic nature of our approach ensures a positive feed-back.

5 Symbolic Constant Propagation

The automatic symbolic simplification implied by our linearization allows us to gain much precision when dealing with complex expressions, without the burden

of designing new abstract domains tailored for them. However, the analysis is still sensitive to program transformations that decompose expressions and introduce new temporary variables—such as common sub-expression elimination or register spilling. In order to be immune to this problem, one must generally use an expressive, and so, costly, *relational* domain. We propose an alternate, lightweight solution based on a kind of constant domain that tracks assignments dynamically and propagate symbolic expressions within transfer functions.

5.1 The Symbolic Constant Domain

Enriched Expressions. We denote by $\mathcal{C}$ the set of all syntactic expressions, enriched with one element $\top^{\mathcal{C}}$ denoting 'any value.' The flat ordering $\sqsubseteq^{\mathcal{C}}$ is defined as $X \sqsubseteq^{\mathcal{C}} Y \iff Y = \top^{\mathcal{C}}$ or $X = Y$. The concrete semantics $[\![\cdot]\!]$ of Fig. 4 is extended to $\mathcal{C}$ as $[\![\top^{\mathcal{C}}]\!](\rho) = \mathbb{I}$. We also use two functions on expression trees: $occ : \mathcal{C} \to \mathcal{P}(\mathcal{V})$ that returns the set of variables occurring in an expressing, and $subst : \mathcal{C} \times \mathcal{V} \times \mathcal{C} \to \mathcal{C}$ that substitutes, in its first argument, every occurrence of a given variable by its last argument. Their definition on non$-\top^{\mathcal{C}}$ elements is quite standard and we do not present it here. They are extended to $\mathcal{C}$ as follows: $occ(\top^{\mathcal{C}}) \stackrel{\text{def}}{=} \emptyset$, $subst(e, V, \top^{\mathcal{C}})$ equals e when $V \notin occ(e)$ and $\top^{\mathcal{C}}$ when $V \in occ(e)$.

Abstract Symbolic Environments. The *symbolic constant domain* is the set $\mathcal{D}^{\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{V} \to \mathcal{C}$ restricted as follows: there must be no cyclic dependencies in a map $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$, that is, pair-wise distinct variables $V_1, \ldots, V_n$ such that $\forall i,\ V_i \in occ(S^{\mathcal{C}}(V_{i+1}))$ and $V_n \in occ(S^{\mathcal{C}}(V_1))$. The partial order $\sqsubseteq^{\mathcal{C}}$ on $\mathcal{D}^{\mathcal{C}}$ is the point-wise extension of that on $\mathcal{C}$. Each element $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$ represents the set of environments compatible with the symbolic information:

Definition 6. $\gamma^{\mathcal{C}}(S^{\mathcal{C}}) \stackrel{\text{def}}{=} \{ \rho \in (\mathcal{V} \to \mathbb{I}) \mid \forall k,\ \rho(V_k) \in [\![S^{\mathcal{C}}(V_k)]\!](\rho) \}$.

Main Theorem. Our approach relies on the fact that applying a substitution from $S^{\mathcal{C}}$ to any expression is sound with respect to $\gamma^{\mathcal{C}}(S^{\mathcal{C}})$:

Theorem 5. $\forall e, V, S^{\mathcal{C}},\ \gamma^{\mathcal{C}}(S^{\mathcal{C}}) \models e \preceq subst(e, V, S^{\mathcal{C}}(V))$.

Abstract Operators. We now define the following operators on $\mathcal{D}^{\mathcal{C}}$:

Definition 7.

- $\{\!| V \leftarrow e |\!\}^{\mathcal{C}}(S^{\mathcal{C}})(V_k) \stackrel{\text{def}}{=} \begin{cases} subst(e,\ V,\ S^{\mathcal{C}}(V)) & \text{if } V = V_k \\ subst(S^{\mathcal{C}}(V_k),\ V,\ S^{\mathcal{C}}(V)) & \text{if } V \neq V_k \end{cases}$
- $\{\!| e \bowtie 0\ ? |\!\}^{\mathcal{C}}(S^{\mathcal{C}}) \stackrel{\text{def}}{=} S^{\mathcal{C}}$,
- $(S^{\mathcal{C}} \cup^{\mathcal{C}} T^{\mathcal{C}})(V_k) \stackrel{\text{def}}{=} \begin{cases} S^{\mathcal{C}}(V_k) & \text{if } S^{\mathcal{C}}(V_k) = T^{\mathcal{C}}(V_k) \\ \top^{\mathcal{C}} & \text{otherwise} \end{cases}$
- $S^{\mathcal{C}} \cap T^{\mathcal{C}} \stackrel{\text{def}}{=} S^{\mathcal{C}}$

Our assignment $V \leftarrow e$ first substitutes V with $S^{\mathcal{C}}(V)$ in $S^{\mathcal{C}}$ and e before adding the information that V maps to the substituted e. This is necessary to remove all prior information on V (no longer valid after the assignment) and prevent the apparition of dependency cycles. As we are only interested in propagating

assignments, tests are abstracted as the identity, which is sound but coarse. Our union abstraction only keeps syntactically equal expressions. This corresponds to the least upper bound with respect to $\sqsubseteq^{\mathcal{C}}$. Our intersection keeps only the information of the left argument. All these operators respect the non-cyclicity condition. Note that one could be tempted to refine the intersection by mixing information from the left and right arguments in order to minimize the number of variables mapping to $\top^{\mathcal{C}}$. Unfortunately, careless mixing may break the non-cyclicity condition. We settled, as a simpler but safe solution, to keeping the left argument. Finally, we do not need any widening: at each abstract iteration, unstable symbolic expressions are directly replaced with $\top^{\mathcal{C}}$ when applying $\cup^{\mathcal{C}}$, and so, become stable.

5.2 Integration with a Numerical Abstract Domain

Given a numerical abstract domain $\mathcal{D}^\sharp$, the domain $\mathcal{D}^{\sharp\times\mathcal{C}}$ is obtained by combining $\mathcal{D}^\sharp_{\mathcal{L}}$ with $\mathcal{D}^{\mathcal{C}}$ the following way:

Definition 8.

- $\mathcal{D}^{\sharp\times\mathcal{C}} \stackrel{\text{def}}{=} \mathcal{D}^\sharp \times \mathcal{D}^{\mathcal{C}}$,
- $\sqsubseteq^{\sharp\times\mathcal{C}}$, $\cup^{\sharp\times\mathcal{C}}$ *and* $\cap^{\sharp\times\mathcal{C}}$ *are defined pair-wise, and* $\nabla^{\sharp\times\mathcal{C}} \stackrel{\text{def}}{=} \nabla^\sharp \times \cup^{\mathcal{C}}$,
- $\gamma^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} \gamma^\sharp(R^\sharp) \cap \gamma^{\mathcal{C}}(S^{\mathcal{C}})$,
- $\{\!| V \leftarrow e |\!\}^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} (\{\!| V \leftarrow \mathit{strat}(e, S^{\mathcal{C}}) |\!\}^\sharp_{\mathcal{L}}(R^\sharp),\ \{\!| V \leftarrow e |\!\}^{\mathcal{C}}(S^{\mathcal{C}}))$
- $\{\!| e \bowtie 0\ ? |\!\}^{\sharp\times\mathcal{C}}(R^\sharp, S^{\mathcal{C}}) \stackrel{\text{def}}{=} (\{\!| \mathit{strat}(e, S^{\mathcal{C}}) \bowtie 0\ ? |\!\}^\sharp_{\mathcal{L}}(R^\sharp),\ \{\!| e \bowtie 0\ ? |\!\}^{\mathcal{C}}(S^{\mathcal{C}}))$

Where $\mathit{strat}(e, S^{\mathcal{C}})$ *is a* substitution strategy *that may perform sequences of substitutions of the form* $f \mapsto \mathit{subst}(f, V, S^{\mathcal{C}}(V))$ *in* e*, for any variables* V*.*

All information in $\mathcal{D}^{\mathcal{C}}$ and $\mathcal{D}^\sharp$ are computed independently, except that the symbolic information is used in the transfer functions for $\mathcal{D}^\sharp_{\mathcal{L}}$. The next section discusses the choice of a strategy *strat*. Note that, although we chose in this presentation to abstract the semantics of Fig. 4, our construction can be used on any class of expressions, including floating-point and non-numerical expressions.

5.3 Substitution Strategies

Any sequence of substitutions extracted from the current symbolic constant information is sound, but some give better results than others. As for the intervalization of Sect. 4.3, we rely on carefully designed strategies.

Full Propagation. Thanks to the non-cyclicity of elements $S^{\mathcal{C}} \in \mathcal{D}^{\mathcal{C}}$, we can safely perform all substitutions $f \mapsto \mathit{subst}(f, V, S^{\mathcal{C}}(V))$ for all V in any order, and reach a normal form. This gives a first basic substitution strategy. However, because our goal is to perform linearization-driven simplifications, it is important to avoid substituting with variable-free expressions or we may lose correlations between multiple occurrences of variables. For instance, full substitution in the assignment $Z \leftarrow X - 0.5 \times Y$ with the environment $S^{\mathcal{C}} = [X \mapsto [0,1],\ Y \mapsto X]$ results in $Z \leftarrow [0,1] - 0.5 \times [0,1]$, and so, $Z \in [-0.5, 1]$. Avoiding variable-free substitutions, this gives $Z \leftarrow X - 0.5 \times X$, and so, $Z \in [0, 0.5]$, which is more

precise. This refined strategy also succeeds in proving that $Y \in [0, 20]$ in the example of Fig. 1 by substituting Y with X in the test $Y \leq 0$.

Enforcing Determinism and Linearity. Non-determinism in expressions is a major source of precision loss. Thus, a strategy is to avoid substituting V with $S^{\mathcal{C}}(V)$ whenever $\#(\llbracket S^{\mathcal{C}}(V) \rrbracket \circ \gamma)(X^\sharp) > 1$. As this property is not easily computed, we propose the following sufficient syntactic criterion: $S^{\mathcal{C}}(V)$ should not be $\top^{\mathcal{C}}$ nor contain a non-singleton interval. This strategy gives the expected result in the example of Fig. 1. Likewise, one may wish to avoid substituting with non-linear expressions, as they must be subsequently intervalized, which is a cause of precision loss. However, disabling too many substitutions may prevent the linearization step to exploit correlations. Suppose that we break the last assignment of Fig. 2 in three parts: $U \leftarrow X \times Y$; $V \leftarrow (1 - X) \times Z$; $T \leftarrow U - V$. Then, the interval domain with linearization and symbolic constant propagation will not be able to prove that $T \in [0, 0.3]$ unless we allow substituting, in T, U and V with their *non-linear* symbolic value.

Gaining More Precision. More precision can be achieved by slightly altering the definition of $\mathcal{D}^{\sharp\times\mathcal{C}}$. A simple but effective idea is to allow several strategies, compute several transfer functions in $\mathcal{D}^\sharp$ in parallel, and take the abstract intersection $\cap^\sharp$ of the results. Another idea is to perform reductions from $\mathcal{D}^{\mathcal{C}}$ to $\mathcal{D}^\sharp$ after each transfer function: $X^\sharp$ is replaced with $\{\!| V_k - S^{\mathcal{C}}(V_k) = 0 \ ? |\!\}^\sharp(X^\sharp)$ for some k. Reductions can be iterated to increase the precision, following Granger's local iterations scheme [10].

6 Application to the Astrée Analyzer

Astrée is an efficient static analyzer focusing on the detection of run-time errors for programs written in a subset of the C programming language, excluding recursion, dynamic memory allocation and concurrent executions. It aims towards a degree of precision sufficient to actually *prove* the absence of run-time errors. This is achieved by specializing the analyzer towards specific program families, introducing various abstract domains, and setting iteration strategy parameters. Currently, the considered family of programs is that of safety, critical, embedded, fly-by-wire avionic software, featuring large reactive loops running for billions of iterations, thousands of global state variables, and pervasive floating-point arithmetics. We refer the reader to [1] for more detailed informations on Astrée.

Integrating the Symbolic Methods. Astrée uses a partially reduced product of several numerical abstract domains, together with both our two symbolic enhancement methods. Relational domains, such as the octagon [12] or digital filtering [9] domains, rely on the linearization to abstract complex floating-point expressions into interval affine forms on reals. The interval domain is refined by combining three versions of each transfer function. Firstly, using the expression unchanged. Secondly, using the linearized expression. Thirdly, applying symbolic constant propagation followed by linearization. We use the simplification-driven

multiplication strategy, as well as the full propagation strategy—not propagating variable-free expressions.

Experimental Results. We present analysis results on a several programs. All the analyses have been carried on an 64-bit AMD Opteron 248 (2 GHz) workstation running Linux, using a single processor. The following table compares the precision and efficiency of Astrée before and after enabling our two symbolic methods:

	without enhancements				**with enhancements**			
code size in lines	analysis time	nb. of iters.	memory	alarms	analysis time	nb. of iters.	memory	alarms
370	1.8s	17	16 MB	0	3.1s	17	16 MB	0
9 500	90s	39	80 MB	8	160s	39	81 MB	8
70 000	2h 40mn	141	559 MB	391	1h 16mn	44	582 MB	**0**
226 000	11h 16mn	150	1.3 GB	141	6h 36mn	86	1.3 GB	1
400 000	22h 9mn	172	2.2 GB	282	13h 52mn	96	2.2 GB	**0**

The precision gain is quite impressive as up to hundreds of alarms are removed. In two cases, this increase in precision is sufficient to achieve zero alarm, which actually *proves* the absence of run-time errors. Moreover, the increase in memory consumption is negligible. Finally, in our largest examples, our enhancement methods *save* analysis time: although each abstract iteration is more costly (up to 25%) this is compensated by the reduced number of iterations needed to stabilize our invariants as a smaller state space is explored.

Discussion. It is possible to use the symbolic constant propagation also in relational domains, but this was not needed in our examples to remove alarms. Our experiments show that, even though the linearization and constant propagation techniques on intervals are not as robust as fully relational abstract domains, they are quite versatile thanks to their parametrization in terms of strategies, and much simpler to implement than even a simple relational abstract domain. Moreover, our methods exhibit a near-linear time and memory cost, which is much more efficient than relational domains.

7 Conclusion

We have proposed, in this article, two techniques, called linearization and symbolic constant propagation, that can be combined together to improve the precision of numerical abstract domains. In particular, we are able to compensate for the lack of non-linear transfer functions in the polyhedron and octagon domains, and for a weak or inexistent level of relationality in the octagon and interval domains. Finally, they help making abstract domains robust against program transformations. Thanks to their parameterization in terms of strategies, they can be finely tuned to take into account semantics as well as syntactic program features. They are also very lightweight in terms of both analysis and development costs. We found out that, in many cases, it is easier and faster to design a

couple of linearization and symbolic propagation strategies to solve a local loss of precision in some program, while keeping the interval abstract domain, than to develop a specific relational abstract domain able to represent the required local properties. Strategies also proved reusable on programs belonging to the same family. Practical results obtained within the ASTRÉE static analyzer show that our methods both increase the precision and save analysis time. They were key in proving the absence of run-time errors in real-life critical embedded avionics software.

Future Work. Because the precision gain strongly depends upon the multiplication strategy used in our linearization and the propagation strategy used in the symbolic constant domain, a natural extension of our work is to try and design new such strategies, adapted to different *practical* cases. A more challenging task would be to provide *theoretical* guarantees that some strategies make abstract domains immune to given classes of program transformations.

Acknowledgments. We would like to thank all the former and present members of the ASTRÉE team: B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, D. Monniaux and X. Rival. We would also like to thank the anonymous referees for their useful comments.

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM PLDI'03*, volume 548030, pages 196–207. ACM Press, 2003.

[2] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *FMPA'93*, volume 735 of *LNCS*, pages 128–14. Springer, 1993.

[3] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *SAS'04*, volume 3148 of *LNCS*, pages 312–327. Springer, 2004.

[4] C. Colby. *Semantics-Based Program Analysis via Symbolic Composition of Transfer Relations.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.

[5] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL'77*, pages 238–252. ACM Press, 1977.

[7] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM POPL'78*, pages 84–97. ACM Press, 1978.

[9] J. Feret. Static analysis of digital filters. In *ESOP'04*, volume 2986 of *LNCS*. Springer, 2004.

[10] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, volume 652 of *LNCS*, pages 68–79. Springer, 1992.

[11] G. Kildall. A unified approach to global program optimization. In *ACM POPL'73*, pages 194–206. ACM Press, 1973.

[12] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, 2001.
[13] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.
[14] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, dec 2004.
[15] A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.
[16] M. Vinícius, A. Andrade, J. L. D. Comba, and J. Stolfi. Affine arithmetic. In *INTERVAL'94*, 1994.

Synthesis of Reactive(1) Designs*

Nir Piterman[1], Amir Pnueli[2], and Yaniv Sa'ar[3]

[1] EPFL - I&C - MTC, 1015, Lausanne, Switzerland
firstname.lastname@epfl.ch
[2] Department of Computer Science, Weizmann Institute of Science, Rehovot, 76100, Israel
firstname.lastname@weizmann.ac.il
[3] Department of Computer Science, Ben Gurion University, Beer-Sheva, Israel
saary@cs.bgu.ac.il

Abstract. We consider the problem of synthesizing digital designs from their LTL specification. In spite of the theoretical double exponential lower bound for the general case, we show that for many expressive specifications of hardware designs the problem can be solved in time N^3, where N is the size of the state space of the design. We describe the context of the problem, as part of the Prosyd European Project which aims to provide a property-based development flow for hardware designs. Within this project, synthesis plays an important role, first in order to check whether a given specification is realizable, and then for synthesizing part of the developed system. The class of LTL formulas considered is that of Generalized Reactivity(1) (generalized Streett(1)) formulas, i.e., formulas of the form:

$$(\square \diamondsuit p_1 \wedge \cdots \wedge \square \diamondsuit p_m) \rightarrow (\square \diamondsuit q_1 \wedge \cdots \wedge \square \diamondsuit q_n)$$

where each p_i, q_i is a boolean combination of atomic propositions. We also consider the more general case in which each p_i, q_i is an arbitrary past LTL formula over atomic propositions.

For this class of formulas, we present an N^3-time algorithm which checks whether such a formula is realizable, i.e., there exists a circuit which satisfies the formula under any set of inputs provided by the environment. In the case that the specification is realizable, the algorithm proceeds to construct an automaton which represents one of the possible implementing circuits. The automaton is computed and presented symbolically.

1 Introduction

One of the most ambitious and challenging problems in reactive systems construction is the automatic synthesis of programs and (digital) designs from logical specifications. First identified as Church's problem [Chu63], several methods have been proposed for its solution ([BL69], [Rab72]). The two prevalent approaches to solving the synthesis problem were by reducing it to the emptiness problem of tree automata, and viewing it as the solution of a two-person game. In these preliminary studies of the problem,

* This research was supported in part by the Israel Science Foundation (grant no.106/02-1), European community project Prosyd, the John von-Neumann Minerva center for Verification of Reactive Systems, NSF grant CCR-0205571, ONR grant N00014-99-1-0131, and SRC grant 2004-TJ-1256.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 364–380, 2006.
© Springer-Verlag Berlin Heidelberg 2006

the logical specification that the synthesized system should satisfy was given as an S1S formula.

This problem has been considered again in [PR89a] in the context of synthesizing reactive modules from a specification given in Linear Temporal Logic (LTL). This followed two previous attempts ([CE81], [MW84]) to synthesize programs from temporal specification which reduced the synthesis problem to satisfiability, ignoring the fact that the environment should be treated as an adversary. The method proposed in [PR89a] for a given LTL specification φ starts by constructing a Büchi automaton $\mathcal{B}_\varphi$, which is then determinized into a deterministic Rabin automaton. This double translation may reach complexity of double exponent in the size of φ. Once the Rabin automaton is obtained, the game can be solved in time $n^{O(k)}$, where n is the number of states of the automaton and k is the number of accepting pairs.

The high complexity established in [PR89a] caused the synthesis process to be identified as hopelessly intractable and discouraged many practitioners from ever attempting to use it for any sizeable system development. Yet there exist several interesting cases where, if the specification of the design to be synthesized is restricted to simpler automata or partial fragments of LTL, it has been shown that the synthesis problem can be solved in polynomial time. Representative cases are the work in [AMPS98] which presents (besides the generalization to real time) efficient polynomial solutions (N^2) to games (and hence synthesis problems) where the acceptance condition is one of the LTL formulas $\Box p$, $\Diamond q$, $\Box\Diamond p$, or $\Diamond\Box q$. A more recent paper is [AT04] which presents efficient synthesis approaches for the LTL fragment consisting of a boolean combinations of formulas of the form $\Box p$.

This paper can be viewed as a generalization of the results of [AMPS98] and [AT04] into the wider class of *generalized Reactivity*(1) formulas (GR(1)), i.e. formulas of the form

$$(\Box\Diamond p_1 \wedge \cdots \wedge \Box\Diamond p_m) \quad \rightarrow \quad (\Box\Diamond q_1 \wedge \cdots \wedge \Box\Diamond q_n) \tag{1}$$

Following the developments in [KPP05], we show how any synthesis problem whose specification is a GR(1) formula can be solved in time N^3, where N is the size of the state space of the design. Furthermore, we present a (symbolic) algorithm for extracting a design (program) which implements the specification. We make an argument that the class of GR(1) formulas is sufficiently expressive to provide complete specifications of many designs.

This work has been developed as part of the Prosyd project (see www.prosyd.org) which aims at the development of a methodology and a tool suit for the property-based construction of digital circuits from their temporal specification. Within the prosyd project, synthesis techniques are applied to check first whether a set of properties is *realizable*, and then to automatically produce digital designs of smaller units.

2 Preliminaries

2.1 Linear Temporal Logic

We assume a countable set of Boolean variables (propositions) $\mathcal{V}$. LTL formulas are constructed as follows.

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

As usual we denote $\neg(\neg\varphi \vee \neg\psi)$ by $\varphi \wedge \psi$, $\text{T}\mathcal{U}\varphi$ by $\diamondsuit\,\varphi$ and $\neg\diamondsuit\neg\varphi$ by $\square\,\varphi$. A formula that does not include temporal operators is a *Boolean formula*.

A model σ for a formula φ is an infinite sequence of truth assignments to propositions. Namely, if P' is the set of propositions appearing in φ, then for every finite set P such that $P' \subseteq P$, a word in $(2^P)^\omega$ is a model. We denote by $\sigma(i)$ the set of propositions at position i, that is $\sigma = \sigma(0), \sigma(1), \ldots$. We present an inductive definition of when a formula holds in model σ at position i.

- For $p \in P$ we have $\sigma, i \models p$ iff $p \in \sigma(i)$.
- $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$
- $\sigma, i \models \varphi \vee \psi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \psi$
- $\sigma, i \models \bigcirc\,\varphi$ iff $\sigma, i+1 \models \varphi$
- $\sigma, i \models \varphi\mathcal{U}\psi$ iff there exists $k \geq i$ such that $\sigma, k \models \psi$ and $\sigma, j \models \varphi$ for all j, $i \leq j < k$

For a formula φ and a position $j \geq 0$ such that $\sigma, j \models \varphi$, we say that φ *holds at position* j of σ. If $\sigma, 0 \models \varphi$ we say that φ *holds* on σ and denote it by $\sigma \models \varphi$. A set of models L satisfies φ, denoted $L \models \varphi$, if every model in L satisfies φ.

We are interested in the question of *realizability* of LTL specifications [PR89b]. Assume two sets of variables $\mathcal{X}$ and $\mathcal{Y}$. Intuitively $\mathcal{X}$ is the set of input variables controlled by the environment and $\mathcal{Y}$ is the set of system variables. With no loss of generality, we assume that all variables are Boolean. Obviously, the more general case that $\mathcal{X}$ and $\mathcal{Y}$ range over arbitrary finite domains can be reduced to the Boolean case. *Realizability* amounts to checking whether there exists an *open controller* that satisfies the specification. Such a controller can be represented as an automaton which, at any step, inputs values of the $\mathcal{X}$ variables and outputs values for the $\mathcal{Y}$ variables. Below we formalize the notion of checking realizability and *synthesis*, namely, the construction of such controllers.

Realizability for LTL specifications is 2EXPTIME-complete [PR90]. We are interested in a subset of LTL for which we solve realizability and synthesis in polynomial time. The specifications we consider are of the form $\varphi = \varphi_e \rightarrow \varphi_s$. We require that φ_α for $\alpha \in \{e, s\}$ can be rewritten as a conjunction of the following parts.

- φ_i^α - a Boolean formula which characterizes the initial states of the implementation.
- φ_t^α - a formula of the form $\bigwedge_{i \in I} \square\, B_i$ where each B_i is a Boolean combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc\, v$ where $v \in \mathcal{X}$ if $\alpha = e$, and $v \in \mathcal{X} \cup \mathcal{Y}$ otherwise.
- φ_g^α - a formula of the form $\bigwedge_{i \in I} \square \diamondsuit\, B_i$ where each B_i is a Boolean formula.

It turns out that most of the specifications written in practice can be rewritten to this format[1]. In Section 7 we discuss also cases where the formulas φ_g^α have also sub-formulas of the form $\square(p \rightarrow \diamondsuit\, q)$ where p and q are Boolean formulas, and additional cases which can be converted to the GR(1) format.

[1] In practice, the specification is usually given in this format. The specification is a collection of assumptions and requirements with the semantics that all assumptions imply all requirements. Every assumption or requirement is usually of a very simple formula similar to the required form.

2.2 Game Structures

We reduce the realizability problem of an LTL formula to the decision of winner in games. We consider two-player games played between a system and an environment. The goal of the system is to satisfy the specification regardless of the actions of the environment. Formally, we have the following.

A *game structure* (GS) $G : \langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi \rangle$ consists of the following components.

- $V = \{u_1, \ldots, u_n\}$: A finite set of typed *state variables* over finite domains. With no loss of generality, we assume they are all Boolean. We define a *state* s to be an interpretation of V, assigning to each variable $u \in V$ a value $s[u] \in \{0, 1\}$. We denote by Σ the set of all states. We extend the evaluation function $s[\cdot]$ to Boolean expressions over V in the usual way. An *assertion* is a Boolean formula over V. A state s satisfies an assertion φ denoted $s \models \varphi$, if $s[\varphi] = \mathbf{true}$. We say that s is a φ-state if $s \models \varphi$.
- $\mathcal{X} \subseteq V$ is a set of *input variables*. These are variables controlled by the environment. Let D_X denote the possible valuations to variables in $\mathcal{X}$.
- $\mathcal{Y} = V \setminus \mathcal{X}$ is a set of *output variables*. These are variables controlled by the system. Let D_Y denote the possible valuations for the variables in $\mathcal{Y}$.
- Θ is the initial condition. This is an assertion characterizing all the initial states of G. A state is called *initial* if it satisfies Θ.
- $\rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ is the transition relation of the environment. This is an assertion, relating a state $s \in \Sigma$ to a possible next input value $\boldsymbol{\xi}' \in D_X$, by referring to unprimed copies of $\mathcal{X}$ and $\mathcal{Y}$ and primed copies of $\mathcal{X}$. The transition relation ρ_e identifies valuation $\boldsymbol{\xi}' \in D_X$ as a possible *input* in state s if $(s, \boldsymbol{\xi}') \models \rho_e(\mathcal{X}, \mathcal{Y}, \mathcal{X}')$ where $(s, \boldsymbol{\xi}')$ is the joint interpretation which interprets $u \in V$ as $s[u]$ and for $v \in \mathcal{X}$ interprets v' as $\boldsymbol{\xi}'[v]$.
- $\rho_s(\mathcal{X}, \mathcal{Y}, \mathcal{X}', \mathcal{Y}')$ is the transition relation of the system. This is an assertion, relating a state $s \in \Sigma$ and an input value $\boldsymbol{\xi}' \in D_X$ to a next output value $\boldsymbol{\eta}' \in D_Y$, by referring to primed and unprimed copies of V. The transition relation ρ_s identifies a valuation $\boldsymbol{\eta}' \in D_Y$ as a possible *output* in state s reading input $\boldsymbol{\xi}'$ if $(s, \boldsymbol{\xi}', \boldsymbol{\eta}') \models \rho_s(V, V')$ where $(s, \boldsymbol{\xi}', \boldsymbol{\eta}')$ is the joint interpretation which interprets $u \in \mathcal{X}$ as $s[u]$, u' as $\boldsymbol{\xi}'[u]$, and similarly for $v \in \mathcal{Y}$.
- φ is the winning condition, given by an LTL formula.

For two states s and s' of G, s' is a *successor* of s if $(s, s') \models \rho_e \wedge \rho_s$. We freely switch between $(s, \boldsymbol{\xi}') \models \rho_e$ and $\rho_e(s, \boldsymbol{\xi}') = 1$ and similarly for ρ_s. A *play* σ of G is a maximal sequence of states $\sigma : s_0, s_1, \ldots$ satisfying *initiality* namely $s_0 \models \Theta$, and *consecution* namely, for each $j \geq 0$, s_{j+1} is a successor of s_j. Let G be an GS and σ be a play of G. From a state s, the environment chooses an input $\boldsymbol{\xi}' \in D_X$ such that $\rho_e(s, \boldsymbol{\xi}') = 1$ and the system chooses an output $\boldsymbol{\eta}' \in D_Y$ such that $\rho_s(s, \boldsymbol{\xi}', \boldsymbol{\eta}') = \rho_s(s, s') = 1$.

A play σ is *winning for the system* if it is infinite and it satisfies φ. Otherwise, σ is *winning for the environment*.

A *strategy* for the system is a partial function $f : \Sigma^+ \times D_X \mapsto D_Y$ such that if $\sigma = s_0, \ldots s_n$ then for every $\boldsymbol{\xi}' \in D_X$ such that $\rho_e(s_n, \boldsymbol{\xi}') = 1$ we have $\rho_s(s_n, \boldsymbol{\xi}', f(\sigma, \boldsymbol{\xi}')) = 1$. Let f be a strategy for the system, and $s_0 \in \Sigma$. A play $s_0, s_1, \ldots$ is said to be *compliant* with strategy f if for all $i \geq 0$ we have $f(s_0, \ldots, s_i, s_{i+1}[\mathcal{X}]) = s_{i+1}[\mathcal{Y}]$, where

$s_{i+1}[\mathcal{X}]$ and $s_{i+1}[\mathcal{Y}]$ are the restrictions of s_{i+1} to variable sets $\mathcal{X}$ and $\mathcal{Y}$, respectively. Strategy f is *winning* for the system from state $s \in \Sigma_G$ if all s-plays (plays departing from s) which are compliant with f are winning for the system. We denote by W_s the set of states from which there exists a winning strategy for the system. A *strategy* for player environment, *winning strategy*, and the *winning set* W_e are defined dually. A GS G is said to be *winning* for the system if all initial states are winning for the system.

Given an LTL specification $\varphi_e \rightarrow \varphi_s$ as explained above and sets of input and output variables $\mathcal{X}$ and $\mathcal{Y}$ we construct a GS as follows. Let $\varphi_\alpha = \varphi_i^\alpha \wedge \varphi_t^\alpha \wedge \varphi_g^\alpha$ for $\alpha \in \{e, s\}$. Then, for Θ we take $\varphi_i^e \wedge \varphi_i^s$. Let $\varphi_t^\alpha = \bigwedge_{i \in I} \square B_i$, then $\rho_\alpha = \bigwedge_{i \in I} \tau(B_i)$, where the translation τ replaces each instance of $\bigcirc v$ by v'. Finally, we set $\varphi = \varphi_g^e \rightarrow \varphi_g^s$. We *solve* the game, attempting to decide whether the game is winning for the environment or the system. If the environment is winning the specification is *unrealizable*. If the system is winning, we *synthesize* a winning strategy which is a *working implementation* for the system as explained in Section 4.

2.3 Fair Discrete Systems

We present implementations as a special case of *fair discrete systems* (FDS) [KP00]. An FDS $\mathcal{D}: \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ consists of the following components.

- $V = \{u_1, ..., u_n\}$: A finite set of Boolean variables. We define a *state* s to be an interpretation of V. Denote by Σ the set of all states. Assertions over V and satisfaction of assertions are defined like in games.
- Θ : The *initial condition*. This is an assertion characterizing all the initial states of the FDS. A state is called *initial* if it satisfies Θ.
- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, relating a state $s \in \Sigma$ to its $\mathcal{D}$-successor $s' \in \Sigma$.
- $\mathcal{J} = \{J_1, \ldots, J_m\}$: A set of *justice requirements* (weak fairness). Each requirement $J \in \mathcal{J}$ is an assertion which is intended to hold infinitely many times in every computation.
- $\mathcal{C} = \{(p_1, q_1), \ldots, (p_n, q_n)\}$: A set of *compassion requirements* (strong fairness). Each requirement $(p, q) \in \mathcal{C}$ consists of a pair of assertions, such that if a computation contains infinitely many p-states, it should also hold infinitely many q-states.

We define a *run* of the FDS $\mathcal{D}$ to be a maximal sequence of states $\sigma : s_0, s_1, ...$, satisfying the requirements of

- *Initiality:* s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution:* For every $j \geq 0$, the state s_{j+1} is a $\mathcal{D}$-successor of the state s_j.

The sequence σ being maximal means that either σ is infinite, or $\sigma = s_0, \ldots, s_k$ and s_k has no $\mathcal{D}$-successor.

A run σ is defined to be a *computation* of $\mathcal{D}$ if it is infinite and satisfies the following additional requirements:

- *Justice:* For each $J \in \mathcal{J}$, σ contains infinitely many J-positions, i.e. positions $j \geq 0$, such that $s_j \models J$.
- *Compassion:* For each $(p, q) \in \mathcal{C}$, if σ contains infinitely many p-positions, it must also contain infinitely many q-positions.

We say that an FDS $\mathcal{D}$ *implements* specification φ if every run of $\mathcal{D}$ is infinite, and every computation of $\mathcal{D}$ satisfies φ. An FDS is said to be *fairness-free* if $\mathcal{J} = \mathcal{C} = \emptyset$. It is called a *just transition system* (JDS) if $\mathcal{C} = \emptyset$.

In general, we use FDS's in order to formalize reactive systems. When we formalize concurrent systems which communicate by shared variables as well as most digital designs, the ensuing formal model is that of a JDS (i.e., compassion-free). Compassion is needed only in the case that the program uses built-in synchronization constructs such as semaphores or synchronous communication.

For every FDS, there exists an LTL formula $\varphi_{\mathcal{D}}$, called the *temporal semantics* of $\mathcal{D}$ which fully characterizes the computations of $\mathcal{D}$. It can be written as:

$$\varphi_{\mathcal{D}}: \quad \Theta \wedge \square(\rho(V, \bigcirc V)) \wedge \bigwedge_{J \in \mathcal{J}} \square \diamondsuit J \wedge \bigwedge_{(p,q) \in \mathcal{C}} (\square \diamondsuit p \rightarrow \square \diamondsuit q)$$

where $\rho(V, \bigcirc V)$ is the formula obtained from $\rho(V, V')$ by replacing each instance of primed variable x' by the LTL formula $\bigcirc x$.
Note that in the case that $\mathcal{D}$ is compassion-free (i.e., it is a JDS), then its temporal semantics has the form

$$\varphi_{\mathcal{D}}: \quad \Theta \wedge \square(\rho(V, \bigcirc V)) \wedge \bigwedge_{J \in \mathcal{J}} \square \diamondsuit J$$

It follows that the class of specifications we consider in this paper, as explained at the end of Subsection 2.1, have the form $\varphi = \varphi_e \rightarrow \varphi_s$ where each φ_α, for $\alpha \in \{e, s\}$, is the temporal semantics of an JDS. Thus, if the specification can be realized by an environment which is a JDS and a system which is a JDS (in particular, if none of them requires compassion for their implementation), then the class of specifications we consider here are as general as necessary. Note in particular, that hardware designs rarely assume compassion (strong fairness) as a built-in construct. Thus, we expect most specifications to be realized by hardware designs to fall in the class of GR(1).

3 μ-Calculus and Games

In [KPP05], we consider the case of GR(1) games (called there *generalized Streett(1)* games). In these games the winning condition is an implication between conjunctions of recurrence formulas ($\square \diamondsuit \varphi$ where φ is a Boolean formula). These are exactly the types of goals in the games we defined in Section 2. We show how to solve such games in cubic time [KPP05]. We re-explain here how to compute the winning regions of each of the players and explain how to use the algorithm to extract a winning strategy. We start with a definition of μ-calculus over game structures. We give the μ-calculus formula that characterizes the set of winning states of the system. We explain how we construct from this μ-calculus formula an algorithm to compute the set of winning states. Finally, by saving intermediate values in the computation, we can construct a winning strategy and synthesize an FDS that implements the goal.

3.1 μ-Calculus over Games Structures

We define μ-calculus [Koz83] over game structures. Let G: $\langle V, \mathcal{X}, \mathcal{Y}, \Theta, \rho_e, \rho_s, \varphi \rangle$ be a GS. For every variable $v \in V$ the formulas v and $\neg v$ are *atomic formulas*. Let $Var =$

$\{X, Y, \ldots\}$ be a set of *relational variables*. The μ-calculus formulas are constructed as follows.

$$\varphi ::= v \mid \neg v \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \diamondsuit \varphi \mid \boxminus \varphi \mid \mu X \varphi \mid \nu X \varphi$$

A formula ψ is interpreted as the set of G-states in Σ in which ψ is true. We write such set of states as $[[\psi]]^e_G$ where G is the GS and $e : Var \rightarrow 2^\Sigma$ is an *environment*. The environment assigns to each relational variable a subset of Σ. We denote by $e[X \leftarrow S]$ the environment such that $e[X \leftarrow S](X) = S$ and $e[X \leftarrow S](Y) = e(Y)$ for $Y \neq X$. The set $[[\psi]]^e_G$ is defined inductively as follows[2].

- $[[v]]^e_G = \{s \in \Sigma \mid s[v] = 1\}$
- $[[\neg v]]^e_G = \{s \in \Sigma \mid s[v] = 0\}$
- $[[X]]^e_G = e(X)$
- $[[\varphi \vee \psi]]^e_G = [[\varphi]]^e_G \cup [[\psi]]^e_G$
- $[[\varphi \wedge \psi]]^e_G = [[\varphi]]^e_G \cap [[\psi]]^e_G$
- $[[\diamondsuit \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \forall \boldsymbol{x}',\ (s, \boldsymbol{x}') \models \rho_e \rightarrow \exists \boldsymbol{y}' \text{ such that } (s, \boldsymbol{x}', \boldsymbol{y}') \models \rho_s \\ \text{and } (\boldsymbol{x}', \boldsymbol{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\diamondsuit \varphi]]^e_G$ if the system can force the play to reach a state in $[[\varphi]]^e_G$. That is, regardless of how the environment moves from s, the system can choose an appropriate move into $[[\varphi]]^e_G$.
- $[[\boxminus \varphi]]^e_G = \left\{ s \in \Sigma \;\middle|\; \begin{array}{l} \exists \boldsymbol{x}' \text{ such that } (s, \boldsymbol{x}') \models \rho_e \text{ and} \\ \forall \boldsymbol{y}',\ (s, \boldsymbol{x}', \boldsymbol{y}') \models \rho_s \rightarrow (\boldsymbol{x}', \boldsymbol{y}') \in [[\varphi]]^e_G \end{array} \right\}$

  A state s is included in $[[\boxminus \varphi]]^e_G$ if the environment can force the play to reach a state in $[[\varphi]]^e_G$. As the environment moves first, it chooses an input $\boldsymbol{x}' \in X$ such that for all choices of the system the successor s is in $[[\varphi]]^e_G$.
- $[[\mu X \varphi]]^e_G = \cup_i S_i$ where $S_0 = \emptyset$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$
- $[[\nu X \varphi]]^e_G = \cap_i S_i$ where $S_0 = \Sigma$ and $S_{i+1} = [[\varphi]]^{e[X \leftarrow S_i]}_G$

When all the variables in φ are bound by either μ or ν the initial environment is not important and we simply write $[[\varphi]]_G$. In case that G is clear from the context we write $[[\varphi]]$.

The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. A μ-calculus formula defines a symbolic algorithm for computing $[[\varphi]]$ [EL86]. For a μ-calculus formula of alternation depth k, the run time of this algorithm is $O(|\Sigma|^k)$. For a full exposition of μ-calculus we refer the reader to [Eme97]. We often abuse notations and write a μ-calculus formula φ instead of the set $[[\varphi]]$.

In some cases, instead of using a very complex formula, it may be more readable to use *vector notation* as in Equation (2) below.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \end{bmatrix} \begin{bmatrix} \mu Y (\diamondsuit Y \ \vee \ p \wedge \diamondsuit Z_2) \\ \mu Y (\diamondsuit Y \ \vee \ q \wedge \diamondsuit Z_1) \end{bmatrix} \tag{2}$$

Such a formula, may be viewed as the mutual fixpoint of the variables Z_1 and Z_2 or equivalently as an equal formula where a single variable Z replaces both Z_1 and Z_2 and ranges over pairs of states [Lic91]. The formula above characterizes the set of states from

[2] Only for finite game structures.

which system can force the game to visit p-states infinitely often and q-states infinitely often. We can characterize the same set of states by the following ‘normal’ formula[3].

$$\varphi = \nu Z\left([\mu Y(\circledcirc Y \ \vee\ p \wedge \circledcirc Z)] \ \wedge\ [\mu Y(\circledcirc Y \ \vee\ q \wedge \circledcirc Z)]\right).$$

3.2 Solving GR(1) Games

Let G be a game where the winning condition is of the following form.

$$\varphi = \bigwedge_{i=1}^{m} \square \diamondsuit J_i^1 \rightarrow \bigwedge_{j=1}^{n} \square \diamondsuit J_j^2$$

Here J_i^1 and J_j^2 are sets of Boolean formulas. In [KPP05] we term these games as generalized Streett(1) games and provide the following μ-calculus formula to solve them. Let $j \oplus 1 = (j \ mod\ n) + 1$.

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ \vdots \\ Z_n \end{bmatrix} \begin{bmatrix} \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_1^2 \wedge \circledcirc Z_2 \ \vee\ \circledcirc Y \ \vee\ \neg J_i^1 \wedge \circledcirc X) \right) \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_2^2 \wedge \circledcirc Z_3 \ \vee\ \circledcirc Y \ \vee\ \neg J_i^1 \wedge \circledcirc X) \right) \\ \vdots \\ \vdots \\ \mu Y \left(\bigvee_{i=1}^{m} \nu X (J_n^2 \wedge \circledcirc Z_1 \ \vee\ \circledcirc Y \ \vee\ \neg J_i^1 \wedge \circledcirc X) \right) \end{bmatrix} \tag{3}$$

Intuitively, for $j \in [1..n]$ and $i \in [1..m]$ the greatest fixpoint $\nu X(J_j^2 \wedge \circledcirc Z_{j\oplus 1} \ \vee\ \circledcirc Y \vee \neg J_i^1 \wedge \circledcirc X)$ characterizes the set of states from which the system can force the play either to stay indefinitely in $\neg J_i^1$ states (thus violating the left hand side of the implication) or in a finite number of steps reach a state in the set $J_j^2 \wedge \circledcirc Z_{j\oplus 1} \vee \circledcirc Y$. The two outer fixpoints make sure that the system wins from the set $J_j^2 \wedge \circledcirc Z_{j\oplus 1} \ \vee\ \circledcirc Y$. The least fixpoint μY makes sure that the unconstrained phase of a play represented by the disjunct $\circledcirc Y$ is finite and ends in a $J_j^2 \wedge \circledcirc Z_{j\oplus 1}$ state. Finally, the greatest fixpoint νZ_j is responsible for ensuring that, after visiting J_j^2, we can loop and visit $J_{j\oplus 1}^2$ and so on. By the cyclic dependence of the outermost greatest fixpoint, either all the sets in J_j^2 are visited or getting stuck in some inner greatest fixpoint, where some J_i^1 is visited only finitely many times.

We include in Fig. 1 a (slightly simplified) code of the implementation of this μ-calculus formula in TLV (see Section 5). We denote J_i^α for $\alpha \in \{1, 2\}$ by $Ji(i, \alpha)$ and $\circledcirc$ by *cox*. We denote conjunction, disjunction, and negation by &, |, and ! respectively. A GreatestFixpoint loop on variable u starts by setting the initial value of u to the set of all states and a LeastFixpoint loop over u starts by setting u to the empty set of states. For both types of fixpoints, the loop terminates if two successive values of u are the same. The greatest fixpoint `GreatestFixpoint(x <= z)`, means that the initial

[3] This does not suggest a canonical translation from vector formulas to plain formulas. The same translation works for the formula in Equation (3) below. Note that the formula in Equation (2) and the formula in Equation (3) have a very similar structure.

```
Func winm(m, n);
  GreatestFixpoint(z)
    For (j in 1...n)
      Let r := 1;
      LeastFixpoint (y)
        Let start := Ji(j,2) & cox(z) | cox(y);
        Let y := 0;
        For (i in 1...m)
          GreatestFixpoint (x <= z)
            Let x := start | !Ji(i,1) & cox(x);
          End -- GreatestFixpoint (x)
          Let x[j][r][i] := x; // store values of x
          Let y := y | x;
        End -- For (i in 1...m)
        Let y[j][r] := y; // store values of y
        Let r := r + 1;
      End -- LeastFixpoint (y)
      Let z := y;
      Let maxr[j] := r - 1;
    End -- For (j in 1...m)
  End -- GreatestFixpoint (z)
  Return z;
End -- Func winm(m, n);
```

Fig. 1. TLV implementation of Equation (3)

value of x is z instead of the universal set of all states. We use the sets $y[j][r]$ and their subsets $x[j][r][i]$ to define n strategies for the system. The strategy f_j is defined on the states in Z_j. We show that the strategy f_j either forces the play to visit J_j^2 and then proceed to $Z_{j\oplus 1}$, or eventually avoid some J_i^1. We show that by combining these strategies, either the system switches strategies infinitely many times and ensures that the play be winning according to right hand side of the implication or eventually uses a fixed strategy ensuring that the play does not satisfy the left hand side of the implication. Essentially, the strategies are "go to $y[j][r]$ for minimal r" until getting to a J_j^2 state and then switch to strategy $j \oplus 1$ or "stay in $x[j][r][i]$".

It follows that we can solve realizability of LTL formulas in the form that interests us in polynomial (cubic) time.

Theorem 1. [KPP05] *Given sets of variables $\mathcal{X}$, $\mathcal{Y}$ whose set of possible valuations is Σ and an* LTL *formula φ with m and n conjuncts, we can determine using a symbolic algorithm whether φ is realizable in time proportional to $(nm|\Sigma|)^3$.*

4 Synthesis

We show how to use the intermediate values in the computation of the fixpoint to produce an FDS that implements φ. The FDS basically follows the strategies explained above.

Let $\mathcal{X}$, $\mathcal{Y}$, and φ be as above. Let G: $\langle V, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \Theta, \varphi_g\rangle$ be the GS defined by $\mathcal{X}$, $\mathcal{Y}$, and φ (where $V = \mathcal{X} \cup \mathcal{Y}$). We construct the following fairness-free FDS. Let

$\mathcal{D}:\langle V_{\mathcal{D}}, \mathcal{X}, \mathcal{Y}_{\mathcal{D}}, \Theta_{\mathcal{D}}, \rho\rangle$ where $V_{\mathcal{D}} = V \cup \{jx\}$ and jx ranges over $[1..n]$, $\mathcal{Y}_{\mathcal{D}} = \mathcal{Y} \cup \{jx\}$, $\Theta_{\mathcal{D}} = \Theta \wedge (jx = 1)$. The variable jx is used to store internally which strategy should be applied. The transition ρ is $\rho_1 \vee \rho_2 \vee \rho_3$ where ρ_1, ρ_2, and ρ_3 are defined as follows.

Transition ρ_1 is the transition taken when a J^2_j state is reached and we change strategy from f_j to $f_{j\oplus 1}$. Accordingly, all the disjuncts in ρ_1 change jx. Transition ρ_2 is the transition taken in the case that we can get closer to a J^2_j state. These transitions go from states in some set $y[j][r]$ to states in the set $y[j][r']$ where $r' < r$. We take care to apply this transition only to states s for which $r > 1$ is the minimal index such that $s \in y[j][r]$. Transition ρ_3 is the transition taken from states $s \in x[j][r][i]$ such that $s \models \neg J^1_i$ and the transition takes us back to states in $x[j][r][i]$. Repeating such a transition forever will also lead to a legitimate computation because it violates the environment requirement of infinitely many visits to J^1_i-states. Again, we take care to apply this transition only to states for which (r, i) are the (lexicographically) minimal indices such that $s \in x[j][r][i]$.

Let $y[j][< r]$ denote the set $\bigcup_{l\in[1..r-1]} y[j][l]$. We write $(r', i') \prec (r, i)$ to denote that the pair (r', i') is lexicographically smaller than the pair (r, i). That is, either $r' < r$ or $r' = r$ and $i' < i$. Let $x[j][\prec(r, i)]$ denote the set $\bigcup_{(r',i')\prec(r,i)} x[j][r'][i']$. The transitions are defined as follows.

$$
\begin{aligned}
\rho_1 \;\;\; &= \bigvee_{j\in[1..n]} (jx{=}j) \wedge z \wedge J^2_j \wedge \rho_e \wedge \rho_s \wedge z' \wedge (jx'{=}j{\oplus}1) \\
\rho_2(j) &= \bigvee_{r>1} y[j][r] \wedge \neg y[j][< r] \wedge \rho_e \wedge \rho_s \wedge y'[j][< r] \\
\rho_2 \;\;\; &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_2(j) \\
\rho_3(j) &= \bigvee_{r} \bigvee_{i\in[1..m]} x[j][r][i] \wedge \neg x[j][\prec(r,i)] \wedge \neg J^1_i \wedge \rho_e \wedge \rho_s \wedge x'[j][r][i] \\
\rho_3 \;\;\; &= \bigvee_{j\in[1..n]} (jx{=}jx'{=}j) \wedge \rho_3(j)
\end{aligned}
$$

The conjuncts $\neg y[j][< r]$ and $\neg x[j][\prec(r, i)]$ appearing in transitions $\rho_2(j)$ and $\rho_3(j)$ ensure the minimality of the indices to which these transitions are respectively applied.

Notice that the above transitions can be computed symbolically. We include below the TLV code that symbolically constructs the transition relation of the synthesized FDS and places it in `trans`. We denote the conjunction of ρ_e and ρ_s by `trans12`.

```
To symb_strategy;
  Let trans := 0;
  For (j in 1...n)
    Let jp1 := (j mod n) + 1;
    Let trans := trans | (jx=j) & z & Ji(j,2) & trans12 &
                         next(z) & (next(jx)=jp1);
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := y[j][1];
```

```
    For (r in 2...maxr[j])
      Let trans := trans | (jx=j) & y[j][r] & !low &
                        trans12 & next(low) & (next(jx)=j);
      Let low := low | y[j][r];
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
  For (j in 1...n)
    Let low := 0;
    For (r in 2...maxr[j])
      For (i in 1...m)
        Let trans := trans | (jx=j) & x[j][r][i] & !low
                                    & !ji(i,1) & trans12 &
                            next(x[j][r][i]) & (next(jx)=j);
        Let low := low | x[j][r][i];
      End -- For (i in 1...m)
    End -- For (r in 2...maxr[j])
  End -- For (j in 1...n)
End -- To symb_strategy;
```

4.1 Minimizing the Strategy

We have created an FDS that implements an LTL goal φ. The set of variables of this FDS includes the given set of input and output variables as well as a 'memory' variable jx. We have quite a liberal policy of choosing the next successor in the case of a visit to J_j^2. We simply choose some successor in the winning set. Here we minimize (symbolically) the resulting FDS. A necessary condition for the soundness of this minimization is that the specification be insensitive to stuttering[4]

Notice, that our FDS is deterministic. For every state and every possible assignment to the variables in $\mathcal{X} \cup \mathcal{Y}$ there exists at most one successor state with this assignment. Thus, removing transitions seems to be of lesser importance. We concentrate on removing redundant states.

As we are using the given sets of variables $\mathcal{X}$ and $\mathcal{Y}$ the only possible candidate states for merging are states that agree on the values of variables in $\mathcal{X} \cup \mathcal{Y}$ and disagree on the value of jx. If we find two states s and s' such that $\rho(s, s')$, $s[\mathcal{X} \cup \mathcal{Y}] = s'[\mathcal{X} \cup \mathcal{Y}]$, and $s'[jx] = s[jx] \oplus 1$, we remove state s. We direct all its incoming arrows to s' and remove its outgoing arrows. Intuitively, we can do that because for every computation that passes through s there exists a computation that stutters once in s (due to the assumption of stuttering insensitivity). This modified computation passes from s to s' and still satisfies all the requirements (we know that stuttering in s is allowed because there exists a transition to s' which agrees with s on all variables).

As mentioned this minimization is performed symbolically. As we discuss in Section 5, it turns out that the minimization actually increases the size of the resulting BDDs.

[4] A specification is insensitive to stuttering if the result of doubling a letter (or replacing a double occurrence by a single occurrence) in a model is still a model. The specifications we consider are allowed to use the next operator, thus they can be sensitive to stuttering. A specification that requires that in some case an immediate response be made would be sensitive to stuttering.

It seems to us that for practical reasons we may want to keep the size of BDDs minimal rather than minimize the automaton. The symbolic implementation of the minimization is given below. The transition *obseq* includes all possible assignments to V and V' such that all variables except jx maintain their values. It is enough to consider the transitions from j to $j\oplus 1$ for all j and then from n to j for all j to remove all redundant states. This is because the original transition just allows to increase jx by one.

```
For (j in 1..n)
  Let nextj := (j mod n)+1;
  reduce(j,nextj);
End -- For (j in 1..n)

For (j in 1..n-1)
  reduce(n,j)
End -- For (j in 1..n-1)

Func reduce(j,k)
  Let idle := trans & obseq & jx=j & next(jx)=k;
  Let states := idle forsome next(V);
  Let add_trans :=
      ((trans & next(states) & next(jx)=j) forsome jx) &
      next(jx)=k;
  Let rem_trans := next(states) & next(jx)=j1 |
                   states & jx=j1;
  Let add_init := ((init & states & jx=j1) forsome jx) &
                  jx=k;
  Let rem_init := states & jx=j;
  Let trans := (trans & !rem_trans) | add_trans;
  Let init := (init & !rem_init) | add_init;
  Return;
End -- Func reduce(j,k)
```

5 Experimental Results

The algorithm described in this paper was implemented within the TLV system [PS96]. TLV is a flexible verification tool implemented at the Weizmann Institute of Science. TLV provides a programming environment which uses BDDs as its basic data type [Bry86]. Deductive and algorithmic verification methods are implemented as procedures written within this environment. We extended TLV's functionality by implementing the algorithms in this paper. We consider two examples. The case of an arbiter and the case of a lift controller.

5.1 Arbiter

We consider the case of an arbiter. Our arbiter has n input lines in which clients request permissions and n output lines in which the clients are granted permission. We

assume that initially the requests are set to zero, once a request has been made it cannot be withdrawn, and that the clients are fair, that is once a grant to a certain client has been given it eventually releases the resource by lowering its request line. Formally, the assumption on the environment in LTL format is below.

$$\bigwedge_i (\overline{r_i} \wedge \Box((r_i{\neq}g_i) \rightarrow (r_i{=}\bigcirc r_i)) \wedge \Box((r_i \wedge g_i) \rightarrow \Diamond \overline{r_i}))$$

We expect the arbiter to initially give no grants, give at most one grant at a time (mutual exclusion), give only requested grants, maintain a grant as long as it is requested, to satisfy (eventually) every request, and to take grants that are no longer needed. Formally, the requirement from the system in LTL format is below.

$$\bigwedge_{i\neq j} \Box\neg(g_i \wedge g_j) \wedge \bigwedge_i \left(\overline{g_i} \wedge \left\{ \begin{array}{ll} \Box((r_i{=}g_i) \ \rightarrow (g_i{=}\bigcirc g_i)) \wedge \\ \Box((r_i \wedge \overline{g_i}) \rightarrow \Diamond g_i) & \wedge \\ \Box((\overline{r_i} \wedge g_i) \rightarrow \Diamond \overline{g_i}) \end{array} \right\} \right)$$

The resulting game is G: $\langle V, \mathcal{X}, \mathcal{Y}, \rho_e, \rho_s, \varphi \rangle$ where

- $\mathcal{X} = \{r_i \mid i = 1, \ldots, n\}$
- $\mathcal{Y} = \{g_i \mid i = 1, \ldots, n\}$
- $\Theta = \bigwedge_i (\overline{r_i} \wedge \overline{g_i})$
- $\rho_e = \bigwedge_i ((r_i{\neq}g_i) \rightarrow (r_i'{=}r_i))$
- $\rho_s = \bigwedge_{i\neq j} \neg(g_i' \wedge g_j') \wedge \bigwedge_i ((r_i{=}g_i) \rightarrow (g_i' = g_i))$
- $\varphi = \bigwedge_i \Box((r_i{\wedge}g_i) \rightarrow \Diamond \overline{r_i}) \rightarrow \bigwedge_i \Box((r_i{\wedge}\overline{g_i}) \rightarrow \Diamond g_i){\wedge}\Box((\overline{r_i}{\wedge}g_i) \rightarrow \Diamond \overline{g_i})$

We simplify φ by replacing $\Box((r_i \wedge g_i) \rightarrow \Diamond \overline{r_i})$ by $\Box \Diamond \neg(r_i \wedge g_i)$ and replacing $\Box((r_i \wedge \overline{g_i}) \rightarrow \Diamond g_i)$ and $\Box((\overline{r_i} \wedge g_i) \rightarrow \Diamond \overline{g_i})$ by $\Box \Diamond (r_i{=}g_i)$. The first simplification is allowed because whenever $r_i \wedge g_i$ holds, the next value of g_i is true. The second simplification is allowed because whenever $r_i \wedge \overline{g_i}$ or $\overline{r_i} \wedge g_i$ holds, the next value of r_i is equal to the current. This results with the simpler goal:

$$\varphi = \bigwedge_i \Box \Diamond \neg(r_i \wedge g_i) \rightarrow \bigwedge_i \Box \Diamond (r_i{=}g_i)$$

In Fig. 2, we present graphs of the run time and size of resulting implementations for the Arbiter example. Implementation sizes are measured in number of BDD nodes.

In Fig. 3 we include the explicit representation of the arbiter for two clients resulting from the application of our algorithm.

5.2 Lift Controller

We consider the case of a lift controller. We build a lift controller for n floors. We assume n button sensors. The lift may be requested on every floor, once the lift has been called on some floor the request cannot be withdrawn. Initially, on all floors there are no requests. Once a request has been fulfilled it is removed. Formally, the assumption on the environment in LTL format is below.

$$\bigwedge_i \left(\overline{b_i} \wedge \Box\left((b_i \wedge f_i) \rightarrow \bigcirc \overline{b_i}\right) \wedge \Box((b_i \wedge \neg f_i) \rightarrow \bigcirc b_i)\right)$$

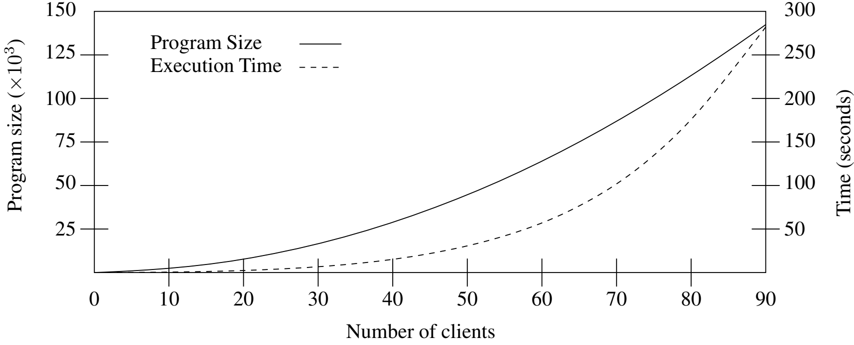


Fig. 2. Running times and program size for the Arbiter example

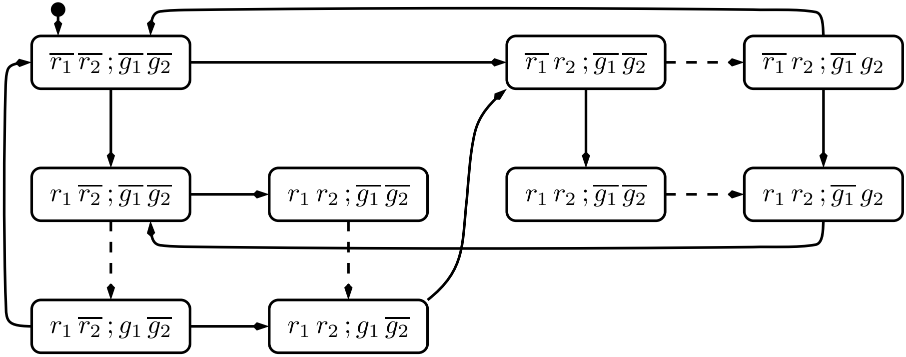


Fig. 3. Arbiter for 2

We expect the lift to initially start on the first floor. We model the location of the lift by an n bit array. Thus we have to demand mutual exclusion on this array. The lift can move at most one floor at a time, and eventually satisfy every request. Formally, the requirement from the system in LTL format is below.

$$\Box(up \rightarrow sb) \;\wedge\; \Box\Diamond(f_1 \vee sb) \;\wedge\; \bigwedge_{i\neq j} \Box\neg(f_i \wedge f_j) \qquad \wedge$$
$$\bigwedge_i \left((i{=}1 \wedge f_i \vee i{\neq}1 \wedge \neg f_i) \wedge \Box\Diamond(b_i \rightarrow f_i) \wedge \Box(f_i \rightarrow \bigcirc(f_i \vee f_{i-1} \vee f_{i+1}))\right)$$

where $up = \bigvee_i (f_i \wedge \bigcirc f_{i+1})$ denotes that the lift moves one floor up, and $sb = \bigvee_i b_i$ denotes that at least one button is pressed. The requirement $\Box(up \rightarrow sb)$ states that the lift should not move up unless some button is pressed. The liveness requirement $\Box\Diamond(f_1 \vee sb)$ states that either some button is pressed infinitely many times, or the lift parks at floor f_1 infinitely many times. Together they imply that when there is no active request, the lift should move down and park at floor f_1.

In Fig. 4 we present graphs of the run time and the size of the resulting implementations for different number of floors. As before, implementation sizes are measured in number of BDD nodes.

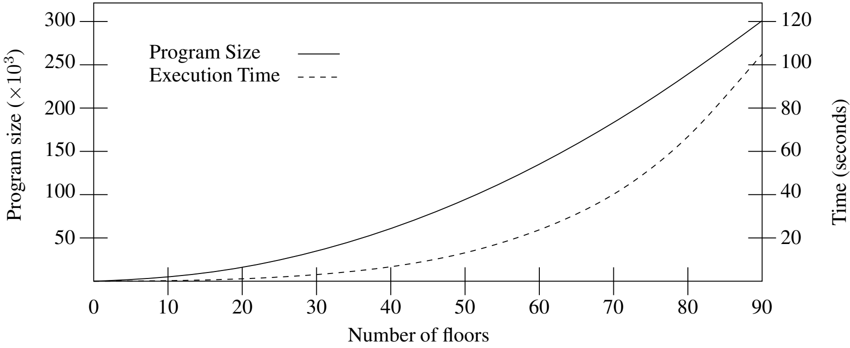


Fig. 4. Running times and program size for the Lift example

6 Extensions

The class of specifications to which the N^3-synthesis algorithm is applicable is wider than the limited form presented in Equation (1). The algorithm can be applied to any specification of the form $(\bigwedge_{i=1}^{m} \varphi_i) \rightarrow (\bigwedge_{j=1}^{n} \psi_j)$, where each φ_i and ψ_j can be specified by an LTL formula of the form $\square \diamondsuit q$ for a past formula q. Equivalently, each φ_i and ψ_j should be specifiable by a deterministic Büchi automaton. This is, for example, the case of the original version of the Arbiter, where the liveness conjuncts were each a response formula of the form $\square(p \rightarrow \diamondsuit q)$.

The way we deal with such a formula is to add to the game additional variables and a transition relation which encodes the deterministic Büchi automaton. For example, to deal with a formula $\square(p \rightarrow \diamondsuit q)$, we add to the game variables a new Boolean variable x with initial condition $x = 1$, and add to the transition relation ρ_e the additional conjunct

$$x' = (q \vee x \wedge \neg p)$$

Table 1. Experiments for Arbiter

N	Recurrence Properties	Response Properties
4	0.05	0.33
6	0.06	0.89
8	0.13	1.77
10	0.25	3.04
12	0.48	4.92
14	0.87	7.30
16	1.16	10.57
18	1.51	15.05
20	1.89	20.70
25	3.03	43.69
30	4.64	88.19
35	6.78	170.50
40	9.50	317.33

We replace in the specification the sub-formula $\Box(p \rightarrow \Diamond q)$ by the conjunct $\Box\Diamond x$. It is not difficult to see that this is a sound transformation. That is, the formula $\Box(p \rightarrow \Diamond q)$ is satisfied by a sequence σ iff there exists an interpretation of the variable x which satisfies the added transition relation and also equals 1 infinitely many times.

Indeed, the in Table 1 we present the performance results of running the Arbiter example with the original specification, to which we applied the above transformation from response to recurrence formulas. The first column presents the results, when the liveness requirements are given as the recurrence formulas $\Box\Diamond(r_i = g_i)$. In the second column, we present the results for the case that we started with the original requirements $\Box(r_i \rightarrow \Diamond)g_i$, and then transformed them into recurrence formulas according to the recipe presented above.

7 Conclusions

We presented an algorithm that solves realizability and synthesis for a subset of LTL. For this subset the algorithm works in cubic time. We also presented an algorithm which reduces the number of states in the synthesized module for the case that the specification is stuttering insensitive.

We have shown that the approach can be applied to wide class of formulas, which covers the full set of generalized reactivity[1] properties. We expect both the system and the environment to be realized by hardware designs. Thus, the temporal semantics of both the system and the environment have a specific form and the implication between the two falls in the set of formulas that we handle. Generalized reactivity[1] certainly covers all the specifications we have so far considered in the Prosyd project. We believe that modifications similar to the ones described in Section 6 would be enough to allow coverage of specifications given in languages such as PSL or FORSPEC [AO04, AFF+02].

Acknowledgments

We thank P. Madhusudan for suggesting that enumerating the states of the controller may be very inefficient.

References

[AFF+02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In 8th *TACAS*, LNCS 2280, 2002.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, 1998.

[AO04] Inc. Accellera Organization. Formal semantics of Accellera(c) property specification language. Appendix B of http://www.eda.org/vfv/docs/PSL-v1.1.pdf, January 2004.

[AT04] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295–311, 1969.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lect. Notes in Comp. Sci.*, pages 52–71. Springer-Verlag, 1981.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25.

[EL86] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional μ-calculus. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 267–278, 1986.

[Eme97] E.A. Emerson. Model checking and the μ-calculus. In N. Immerman and Ph.G. Kolaitis, editors, *Descriptive Complexity and Finite Models*, pages 185–214. AMS, 1997.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Inf. and Comp.*, 163:203–243, 2000.

[KPP05] Y. Kesten, N. Piterman, and A. Pnueli. Bridging the gap between fair simulation and trace inclusion. *Inf. and Comp.*, 200(1):36–61, 2005.

[Lic91] O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, Weizmann Institute of Science, 1991.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Prog. Lang. Sys.*, 6:68–93, 1984.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symp. Princ. of Prog. Lang.*, pages 179–190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. 16th Int. Colloq. Aut. Lang. Prog.*, volume 372 of *Lect. Notes in Comp. Sci.*, pages 652–671. Springer-Verlag, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. 31st IEEE Symp. Found. of Comp. Sci.*, pages 746–757, 1990.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In *Proc.* 8th *Intl. Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, pages 184–195, 1996.

[Rab72] M.O. Rabin. *Automata on Infinite Objects and Churc's Problem*, volume 13 of *Regional Conference Series in Mathematics*. Amer. Math. Soc., 1972.

Systematic Construction of Abstractions for Model-Checking

Arie Gurfinkel, Ou Wei, and Marsha Chechik

Department of Computer Science, University of Toronto
{arie, owei, chechik}@cs.toronto.edu

Abstract. This paper describes a framework, based on Abstract Interpretation, for creating abstractions for model-checking. Specifically, we study how to abstract models of μ-calculus and systematically derive abstractions that are constructive, sound, and precise, and apply them to abstracting Kripke structures. The overall approach is based on the use of bilattices to represent partial and inconsistent information.

1 Introduction

Abstraction plays a fundamental role in combating state-space explosion in model-checking. The goal of abstraction is to construct an abstract model of a system which is small enough to be effectively analyzed, and yet rich enough to yield conclusive results. Success of current abstraction projects, such as SLAM [2] and Bandera [6], indicates that abstraction is an effective technique for enabling model-checking of realistic software systems.

In model-checking, a notion of abstracting a transition system is typically developed as follows: (1) An abstract statespace is defined such that each abstract state corresponds to a set of concrete states. This correspondence can be arbitrary, as in predicate abstraction [17], or influenced by the concrete statespace, as in symmetry reduction [12]. (2) An abstract transition system is constructed by defining a transition relation over this abstract statespace. (3) Finally, the resulting system is argued to be sound, i.e., it is shown to preserve a fragment of the desired temporal logic.

The problem with the above approach is that it is not algorithmic: the techniques used to construct the abstract systems require a certain amount of intuition of users, and extra effort is needed to show that the resulting abstraction is correct. This makes it difficult to understand a specific abstraction method and improve on it. For example, given an abstraction that preserves universal CTL, how should it be changed to preserve the entire CTL? It is also difficult to understand the relationship between different abstract methods. For example, as shown in [25], predicate abstraction and symmetry reduction differ only in their choice of abstract states. However, this insight was not apparent just from the description of these methods.

Given the role abstraction plays in the model-checking process, we believe it is essential to create a general methodology for systematically constructing and analyzing abstractions. In the context of static analysis of programs, such a framework, called Abstract Interpretation (AI), has already been proposed by [7]. It provides a collection of

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 381–397, 2006.
© Springer-Verlag Berlin Heidelberg 2006

notations and tools to formalize the approximation of program semantics, as well as to design and analyze program abstractions. The goal of this paper is to specialize the AI framework to model-checking.

There are a number of ways to do this specialization, given the breadth of model-checking approaches. Our goal here is to create abstractions that preserve properties expressed in the modal μ-calculus [20] (L_μ). Following the recipes of AI, we systematically derive conditions under which an abstract L_μ model is the best abstraction of a concrete one. We guarantee that these abstract models are (a) sound, i.e., if an L_μ formula is satisfied in the abstract model it is satisfied in the concrete, (b) most precise, i.e., satisfy the most properties, and (c) have the desired structural characteristics, e.g., a requirement that an abstraction of a transition system is a transition system as well. These conditions are constructive and, as we show in this paper, can be derived almost mechanically. The algorithm for building a desired abstraction follows from these conditions directly.

The logic L_μ includes negation, so that an L_μ formula $\neg\varphi$ is satisfied iff φ is refuted. If we assume that every formula is either satisfied or refuted in an abstraction as well, it may seem that preserving soundness for *all* L_μ formulas means that such an abstraction must satisfy and refute exactly the same properties as the corresponding concrete model (resulting in a bisimilar model). If the goal is to save space for model-checking, this abstraction would be very limited. Thus, most existing abstractions are restricted to fragments of L_μ, i.e., only to the universal or only to the existential properties (see, e.g., [22]).

The insight we use in this paper is that an abstraction is inherently incomplete: some formulas may be neither satisfied nor refuted by it. We propose to treat satisfaction and refutation independently. If we classify all L_μ formulas using a pair $\langle \mathit{Sat}, \mathit{Ref} \rangle$, where *Sat* contains all the formulas satisfied in an abstract model, while *Ref* contains all the refuted ones, then *Sat* and *Ref* are not necessarily complements of each other. In fact, *Sat* and *Ref* may not even be disjoint, allowing some formulas to be both satisfied and refuted.

Associating knowledge about truth and falsity of every piece of evidence can be naturally encoded using 4-valued Belnap logic [3] which enjoys nice mathematical properties associated with *bilattices* [15, 14]. That is, bilattices enable a uniform approach for handling partial and inconsistent information, allowing reasoning about truth and knowledge in a single theoretical framework. In this paper, by combining the theories of AI with that of bilattices, we obtain a simple and elegant framework for deriving abstractions for L_μ. Due to the generality of bilattices, our results apply not only to the traditional two-valued interpretation of L_μ, but also to its multi-valued [4] and quantitative [10] interpretations.

The contribution of this paper is a general technique, based on AI, for deriving abstractions for model-checking. It allows understanding and comparing different techniques, and provides a methodology for proving soundness and precision of the desired abstraction. We then study this technique on two additional levels. First, we apply it to L_μ models, and then specialize it to abstracting transition systems represented as Kripke structures.

The rest of this paper is organized as follows: after providing the necessary background in Section 2, we show how to lift abstraction between elements to abstraction between sets of elements in Section 3. This gives us a general framework for approximating interpretations of L_μ. In Section 4, we derive abstractions for model-theoretic interpretations of L_μ, and then apply our technique to abstracting transition systems in Section 5. In Section 6, we specialize the results of Section 5 to *boolean* transition systems, and compare them to those obtained by Dams et al. [8]. We relate our technique with other abstraction approaches in Section 7 and summarize our contributions in Section 8.

2 Background

In this section, we introduce the basic concepts of lattice theory, modal μ-calculus, and abstract interpretation.

2.1 Lattices and Monotone Functions

A *lattice* is a partially ordered set $\mathcal{L} = (L, \leq)$ in which every subset B of L has a least upper bound, called *join* and denoted $\sqcup B$, and a greatest lower bound, called *meet* and denoted $\sqcap B$ [9]. A lattice is *distributive* if meet and join distribute over each other, i.e., $(a \sqcap b) \sqcup c = (a \sqcup c) \sqcap (b \sqcup c)$, and $(a \sqcup b) \sqcap c = (a \sqcap c) \sqcup (b \sqcap c)$.

A *De Morgan algebra* is a structure $\mathcal{D} = (L, \leq, -)$, where $(L, \leq)$ is a distributive lattice and $- : L \to L$ is a *negation* that satisfies involution ($- - a = a$) and De Morgan laws: $-(a \sqcap b) = -a \sqcup -b$, and $-(a \sqcup b) = -a \sqcap -b$.

We denote the space of functions from A to B by $A \to B$, or B^A. For example, both $A \to [B \to C]$ and $(C^B)^A$ denote the space of functions from A to functions from B to C.

Let A be a set and $\mathcal{L} = (L, \leq)$ be a lattice. The ordering and operations of $\mathcal{L}$ extend pointwise to L^A, i.e., $f \leq g \Leftrightarrow \forall a \in A \cdot f(a) \leq g(a)$. This turns L^A into a lattice with the same properties as $\mathcal{L}$. In particular, if $\mathcal{L}$ is distributive or De Morgan, so is L^A.

A function f between two partially ordered sets $(A, \leq)$ and $(B, \sqsubseteq)$ is *monotone* (or, *order-preserving*) iff $a \leq b \Rightarrow f(a) \sqsubseteq f(b)$, and *anti-monotone* iff $a \leq b \Rightarrow f(b) \sqsubseteq f(a)$. We use upward ($\uparrow$) and downward ($\downarrow$) arrows to indicate monotone and anti-monotone functions, respectively. For example, $[A \to B]_\uparrow$ denotes the space of all monotone functions from A to B, and $(B^A)_\downarrow$ denotes the space of all anti-monotone functions. Monotone and anti-monotone functions are closed under pointwise meet and join; thus, if B is a lattice, then so are $[A \to B]_\uparrow$ and $[A \to B]_\downarrow$.

2.2 Truth Domains and Sets

A *truth-domain* $\mathcal{D}$ is a collection of elements D, referred to as *truth values*, together with a truth ordering $\sqsubseteq$ and a negation operator $\neg : D \to D$, such that $\mathcal{D} = (D, \sqsubseteq, \neg)$ is a De Morgan algebra. The truth ordering orders the elements based on their truth content; thus, $a \sqsubseteq b$ stands for "a is less true than b". The meet ($\wedge$) and join ($\vee$) of the truth ordering are called *conjunction* and *disjunction*, respectively.

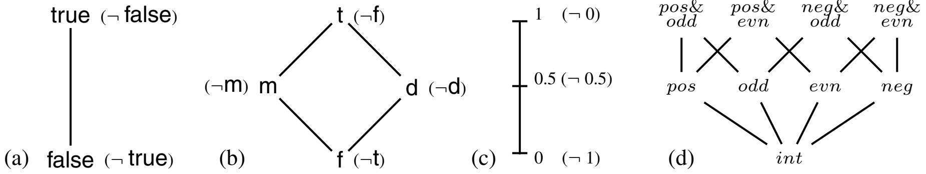


Fig. 1. (a)-(c) Truth domains: (a) 2-valued boolean logic, (b) Belnap logic, and (c) Fuzzy logic. (d) An abstract domain for $\mathbb{Z}$.

The best known truth domain is the classical boolean logic **2** with values true and false. Its truth ordering is shown in the Hasse diagram in Figure 1(a), with negation indicated in parentheses. Other examples include Belnap logic $\mathcal{B}$, shown in Figure 1(b), which extends boolean logic with two additional values: m and d, to represent "unknown" and "inconsistent", respectively; and Fuzzy logic $\mathcal{F}$, shown in Figure 1(c). The truth values of $\mathcal{F}$ are formed by the set of all real numbers in the closed interval $[0, 1]$, where 0 stands for false, 1 for true, and the remaining values stand for degrees of truth; furthermore, negation is defined as $\neg x \triangleq 1 - x$, so that $\neg 0 = 1$ and $\neg 1 = 0$.

Given a collection of elements U, a *set* over U is a function from U to a truth domain. Thus, a boolean (or classical) set is a function from U to **2**, a Belnap set is a function from U to $\mathcal{B}$, and a fuzzy set is a function from U to $\mathcal{F}$. Set ordering and operations are defined by pointwise extensions. Let $S_1, S_2 : U \to D$ be two sets. Then

$$\begin{array}{llll} S_1 \subseteq S_2 & \triangleq \forall x \cdot S_1(x) \sqsubseteq S_2(x) & \quad S_1 \cup S_2 & \triangleq \lambda x \cdot S_1(x) \vee S_2(x) \\ \overline{S_1} & \triangleq \lambda x \cdot \neg S_1(x) & \quad S_1 \cap S_2 & \triangleq \lambda x \cdot S_1(x) \wedge S_2(x). \end{array}$$

2.3 Modal μ-Calculus

In this section, we describe the modal μ-calculus [20], or L_μ.

Definition 1. *Let* Var *be a set of variables and AP be a set of atomic propositions. The logic $L_\mu(AP)$ is the set of all formulas satisfying the grammar*

$$\varphi ::= p \mid z \mid \neg\varphi \mid \varphi \wedge \varphi \mid \Diamond\varphi \mid \mu z \cdot \varphi(z),$$

where $p \in AP$, and $z \in$ Var. Furthermore, z in $\mu z \cdot \varphi(z)$ must occur under the scope of an even number of negations in $\varphi(z)$.

Additionally, we define the following syntactic abbreviations:

$$\varphi \vee \psi \triangleq \neg(\neg\varphi \wedge \neg\psi) \quad \varphi \Rightarrow \psi \triangleq \neg\varphi \vee \psi \quad \Box\varphi \triangleq \neg\Diamond\neg\varphi \quad \nu Z \cdot \varphi(Z) \triangleq \neg\mu Z \cdot \neg\varphi(\neg Z)$$

The modal operator $\Diamond$ is typically interpreted as "an existence of an immediate future". For example, "p" means that p holds now, "$\Diamond p$" means that there exists an immediate future where p holds, and "$\Box p$" means that p holds in all immediate futures. The quantifiers μ and ν stand for least and greatest fixpoint, respectively.

An occurrence of a variable z in a formula φ is *bound* if it appears in the scope of a μ quantifier and is *free* otherwise. For example, z is free in $p \vee \Diamond z$, and is bound in $\mu z \cdot p \vee \Diamond z$. A formula φ is *closed* if it does not contain any free variables.

A *set-based interpretation* of L_μ over a set domain $\mathcal{D}^C$ is a mapping $||\cdot||$ from closed L_μ formulas to $\mathcal{D}$-sets over C. The elements of C are often called *states*, and $||\varphi||(c) = v$ is read as "the degree to which φ is satisfied by (or, true in) a state c is v".

An L_μ *model* is a structure $\mathcal{M} = (\mathcal{D}^C, (p^\mathcal{M})_{p \in AP}, \Diamond^\mathcal{M})$, where $\mathcal{D}^C$ is a set domain; for each $p \in AP$, $p^\mathcal{M}$ is in $\mathcal{D}^C$; and $\Diamond^\mathcal{M} : \mathcal{D}^C \to \mathcal{D}^C$ is a $\subseteq$-monotone function. The set domain is called the *universe* of $\mathcal{M}$, and $p^\mathcal{M}$ and $\Diamond^\mathcal{M}$ are interpretations of atomic propositions and the $\Diamond$ operator, respectively. A model $\mathcal{M}$ gives rise to an L_μ interpretation $||\cdot||^\mathcal{M}$.

The interpretation $||\varphi||^\mathcal{M}_\sigma$ is defined inductively on the structure of the formula φ, where $\sigma : \mathsf{Var} \to \mathcal{D}^C$ is an object assignment for free variables:

$$
\begin{aligned}
||p||^\mathcal{M}_\sigma &\triangleq p^\mathcal{M} & ||z||^\mathcal{M}_\sigma &\triangleq \sigma(z) \\
||\varphi \wedge \psi||^\mathcal{M}_\sigma &\triangleq ||\varphi||^\mathcal{M}_\sigma \cap ||\psi||^\mathcal{M}_\sigma & ||\neg\varphi||^\mathcal{M}_\sigma &\triangleq \overline{||\varphi||^\mathcal{M}_\sigma} \\
||\mu x \cdot \varphi||^\mathcal{M}_\sigma &\triangleq \mathrm{lfp}^\subseteq \left(\lambda S \cdot ||\varphi||^\mathcal{M}_{\sigma[x \mapsto S]}\right) & ||\Diamond\varphi||^\mathcal{M}_\sigma &\triangleq \Diamond^\mathcal{M}(||\varphi||^\mathcal{M}_\sigma)
\end{aligned}
$$

where $\mathrm{lfp}^\subseteq f$ is the $\subseteq$-least fixpoint of f. For a closed L_μ formula φ, $||\varphi||^\mathcal{M}_\sigma = ||\varphi||^\mathcal{M}_{\sigma'}$ for any σ and σ'. Thus, we write $||\varphi||^\mathcal{M}$ for that value, and define it to be the *model-based interpretation* of φ.

Formulas of L_μ are often interpreted over Kripke structures. A *Kripke structure* is a tuple $\mathcal{K} = (AP, C, \mathcal{D}, I, R)$, where AP is a collection of atomic propositions, C is a collection of elements (called *states*), $\mathcal{D}$ is a truth domain, $I : AP \to \mathcal{D}^C$ is a mapping from atomic propositions to sets over C, and $R : C \to \mathcal{D}^C$ is a transition function mapping each state to a set of its successors. For a transition function R, we define a pre-image operator $pre[R] : \mathcal{D}^C \to \mathcal{D}^C$ and its dual $\tilde{pre}[R]$ as:

$$
pre[R](Q)(s) \triangleq \vee_{t \in C}(R(s) \cap Q)(t) \qquad \tilde{pre}[R](Q) \triangleq \overline{pre[R](\overline{Q})}
$$

Intuitively, $pre[R](Q)(s)$ is a degree to which the set $R(s)$ of successors of s has a non-empty intersection with Q. A Kripke structure $\mathcal{K} = (AP, C, \mathcal{D}, I, R)$ gives rise to an $L_\mu(AP)$ model $\mathcal{M}(\mathcal{K}) = (\mathcal{D}^C, (p^{\mathcal{M}(\mathcal{K})})_{p \in AP}, \Diamond^{\mathcal{M}(\mathcal{K})})$, where $p^{\mathcal{M}(\mathcal{K})} \triangleq I(p)$, and $\Diamond^{\mathcal{M}(\mathcal{K})} \triangleq pre[R]$. Finally, the interpretation $||\cdot||^\mathcal{K}$ of L_μ in $\mathcal{K}$ is defined as $||\varphi||^\mathcal{K} \triangleq ||\varphi||^{\mathcal{M}(\mathcal{K})}$.

2.4 Abstract Interpretation

The framework of Abstract Interpretation (AI) provides a collection of tools for systematic design and analysis of semantic approximations [7]. The framework is very flexible and can be applied in various ways. Below, we give a brief overview of AI, summarizing the results used in our work.

Basics of Abstract Interpretation. Inputs to an AI framework are collections of concrete elements C and abstract elements A, called a *concrete* and an *abstract* domain, respectively. A notion of approximation, or abstraction, is formalized by a *soundness relation* $\rho \subseteq A \times C$, where $a \,\rho\, c$ is read as "a ρ-approximates c".

A *concretization function* $\gamma : A \to 2^C$ maps each abstract element to a set of concrete elements corresponding to it: $\gamma(a) \triangleq \{c \mid a \,\rho\, c\}$. An abstract element a is called *consistent* if $\gamma(a) \neq \emptyset$; otherwise, we say a is *inconsistent*. The elements of A can be

thought of as properties, such as “positive” or “odd”, and $\gamma(a)$ as a collection of concrete elements satisfying a. The concretization γ induces an *approximation ordering* $\preceq_\rho$ on A such that $a \preceq_\rho b \Leftrightarrow \gamma(a) \supseteq \gamma(b)$.

Intuitively, $a \preceq_\rho b$ means that a approximates more concrete elements than b; therefore, a is less informative, or equivalently, less precise than b. When viewed as a property, a is weaker than b. For example, knowing that an element is “positive” is less informative than knowing that it is both “positive” and “odd”.

In this paper, an abstract domain A is equipped with an *information ordering* $\preceq_A$ such that $(A, \preceq_A)$ is a lattice and $a \preceq_A b \Rightarrow a \preceq_\rho b$. Thus, we can study properties of an abstract domain independently of any particular soundness relation. Furthermore, we assume that A satisfies “the existence of a best approximation” [7], that is:

$$\forall c \in C \cdot \exists a \in A \cdot (a \,\rho\, c \wedge \forall a' \in A \cdot a' \,\rho\, c \Rightarrow \gamma(a') \supseteq \gamma(a))$$

and use $\alpha : C \to A$ to denote an *abstraction function* that maps each concrete element to its best approximation. Note that for a given c, A may have several best approximations; thus, α is not uniquely defined. In such cases, it is convenient to use the $\preceq_A$-largest α, so that ρ and γ can be expressed as $a \,\rho\, c \Leftrightarrow a \preceq_A \alpha(c)$ and $c \in \gamma(a) \Leftrightarrow a \preceq_A \alpha(c)$, respectively.

A lower bound with respect to $\preceq_\rho$ is called *widening* and is denoted by $\bigtriangledown$. Intuitively, for a set $Q \subseteq A$, $\bigtriangledown Q$ is an abstract element representing the information common to all elements of Q, i.e., $\gamma(\bigtriangledown Q) \supseteq \cup_{q \in Q} \gamma(q)$. In particular, the greatest lower bound $\sqcap_A$ of $\preceq_A$ is a widening. A widening $\bigtriangledown$ is *info-preserving* if for any Q containing no inconsistent elements, $\bigtriangledown Q$ is the best representation of information common to all elements of Q, i.e., $\forall a \in A \cdot \gamma(a) \supseteq \cup_{q \in Q} \gamma(q) \Rightarrow \gamma(a) \supseteq \gamma(\bigtriangledown Q)$.

Abstract domains $(A_1, \preceq_1)$ and $(A_2, \preceq_2)$ are *informationally equivalent* if they represent the same degrees of information, that is, $\forall a_1 \in A_1 \cdot \exists a_2 \in A_2 \cdot \gamma_1(a_1) = \gamma_2(a_2)$ and $\forall a_2 \in A_2 \cdot \exists a_1 \in A_1 \cdot \gamma_1(a_1) = \gamma_2(a_2)$.

For examples in this paper, we use the set of integers $\mathbb{Z}$ as a concrete domain, and the domain $\mathbb{A}$, shown in Figure 1(d), as the abstract domain. The soundness relation $\rho_e \subseteq \mathbb{A} \times \mathbb{Z}$ is self-explanatory, e.g., 2 is ρ_e-approximated by $pos\&evn$, evn, pos, and int, where $pos\&evn$ is its best abstract approximation. Similarly, $\gamma_e(evn)$ is the set *EVEN* of all even numbers, $\gamma_e(neg)$ is the set *NEG* of all negative numbers, $\gamma_e(neg\&evn)$ is *NEG* $\cap$ *EVEN*, etc.

Functional Abstraction. In practice, it is common to synthesize abstractions of complex structures using abstractions of their parts. A particular application is abstraction of functions, or *functional abstraction* [7].

Let $\mathcal{A} = A_1 \to A_2$ and $\mathcal{C} = C_1 \to C_2$ be collections of abstract and concrete functions, where A_1 and A_2 are abstract domains approximating C_1 and C_2, respectively. A soundness relation $\rho_f \subseteq \mathcal{A} \times \mathcal{C}$ is *functional* if g ρ_f-approximates f iff g preserves soundness of f. Formally, ρ_f satisfies

$$g \,\rho_f\, f \Leftrightarrow \forall a_1 \in A_1 \cdot \forall c_1 \in \gamma_1(a_1) \cdot g(a_1) \,\rho_2\, f(c_1) \qquad \text{(functional soundness)}$$

Let $\bigtriangledown$ be a widening operator of A_2, and $\alpha_\bigtriangledown : \mathcal{C} \to \mathcal{A}$ be defined as

$$\alpha_\bigtriangledown(f)(a) \triangleq \bigtriangledown_{c \in \gamma_1(a)} \alpha_2(f(c)) \qquad \text{(functional abstraction)}$$

Then $\alpha_\bigtriangledown(f)$ is a ρ_f-approximation of f, and its precision is determined by the precision of the widening operator used.

Theorem 1. *[7] Let $\mathcal{A}$, $\mathcal{C}$, ρ_f, and α_∇ be as above. If ∇ is info-preserving, then $\alpha_\nabla(f)$ is the best ρ_f-approximation of f.*

One of the main results of AI is that α_∇ preserves fixpoints:

Theorem 2. *[7] Let $(C, \sqsubseteq_C)$ be a lattice, $f : [C \to C]_\uparrow$ be a monotone function, and $(A, \sqsubseteq_A)$ be a lattice approximating C via ρ_C. If the join operator $\vee_A$ of A preserves soundness, i.e., $(\alpha_C(c_1) \vee_A \alpha_C(c_2)) \preceq_A \alpha_C(c_1 \vee_C c_2)$, then the least fixpoint of $\alpha_\nabla(f)$ ρ_C-approximates the least fixpoint of f: $\text{lfp}^{\sqsubseteq_A} \alpha_\nabla(f) \;\rho_C\; \text{lfp}^{\sqsubseteq_C} f$.*

Functional Abstraction and Monotone Functions. Let $\mathcal{A} = [A_1 \to A_2]$ be as above, and assume that A_1 and A_2 are equipped with information orderings $\preceq_1$ and $\preceq_2$, respectively. Then the set $\mathcal{A}_\uparrow = [A_1 \to A_2]_\uparrow$ of $\preceq$-monotone functions is informationally equivalent to $\mathcal{A}$.Furthermore, if ∇ is an info-preserving widening of A_2, then its pointwise extension to functions is also an info-preserving widening of $\mathcal{A}_\uparrow$ [25]. Therefore, we always restrict the abstract domain of functional abstraction to $\preceq$-monotone functions.

3 Abstract Sets

Sets play the role of basic blocks in the definition of L_μ semantics. In this section, we develop an abstraction of sets that preserves all set operations, including set complement. This abstraction gives us the necessary tools for abstracting L_μ models, which we do in Section 4. But it is independent of L_μ and can be used anywhere abstract sets are required.

We assume that C and A are a concrete and an abstract domain, respectively, related by a soundness relation ρ_e and an abstraction function α_e. We aim to lift ρ_e to a soundness relation ρ_s between concrete sets, i.e., functions from C into a fixed truth domain $\mathcal{D}$, and abstract sets, i.e., functions from A into some truth domain $\mathcal{B}$ (potentially different from $\mathcal{D}$). The goal of ρ_s is to preserve set membership: that is, if S_α ρ_s-approximates S, then if $a \in A$ ρ_e-approximates c, $S_\alpha(a)$ must approximate $S(c)$. As always, we also want to know when S_α is a best approximation of a given set S.

We view sets as functions, so it is natural to express ρ_s as a functional abstraction. For this, we must first identify the notion of an abstract truth domain $\mathcal{B}$ and settle on the meaning of "approximating truth values".

3.1 Bilattices as Abstract Truth Domains

Intuitively, an abstract truth-domain $\mathcal{B}$ is a truth-domain and, therefore, has a truth ordering and a negation. It is also an abstract domain and needs an information ordering. Furthermore, truth operations should not interfere with the information ordering. For example, if a and b are in $\mathcal{B}$ and a is less informative than b, then negation of a ($\neg a$) must be less informative than $\neg b$.

A structure that captures our intuition is that of a bilattice, which has been introduced by Ginsberg [15] to enable reasoning with partiality and inconsistency. Here, we briefly describe distributive bilattices.

Definition 2. *[15] A* distributive bilattice *is a structure $\mathcal{B} = (B, \preceq, \sqsubseteq, \neg)$ such that: (1) $\mathcal{B}_i = (B, \preceq)$ is a lattice and $\mathcal{B}_t = (B, \sqsubseteq, \neg)$ is a De Morgan algebra; (2) meet ($\sqcap$) and join ($\sqcup$) of $\mathcal{B}_i$, and meet ($\wedge$) and join ($\vee$) of $\mathcal{B}_t$ are monotone with respect to both $\preceq$ and $\sqsubseteq$; (3) all meets and joins distribute over each other; and (4) negation ($\neg$) is $\preceq$-monotone.*

The ordering $\preceq$ ranks elements of $\mathcal{B}$ with respect to information, and $\sqsubseteq$ ranks them with respect to truth. Operations $\wedge$ and $\vee$ of $\mathcal{B}_t$ are called conjunction and disjunction. In the spirit of AI, we refer to $\sqcap$ and $\sqcup$ as *widening* and *narrowing*, respectively.

De Morgan algebras have a natural connection to bilattices.

Theorem 3. *[15, 14]. Let $\mathcal{D} = (D, \leq, -)$ be a De Morgan algebra, and $\mathcal{B}(\mathcal{D})$ be a structure $(D \times D, \preceq, \sqsubseteq, \neg)$ such that*

$$\langle a, b\rangle \preceq \langle c, d\rangle \triangleq a \leq c \wedge b \leq d \qquad \langle a, b\rangle \sqsubseteq \langle c, d\rangle \triangleq a \leq c \wedge d \leq b \qquad \neg\langle a, b\rangle \triangleq \langle b, a\rangle$$

Then, $\mathcal{B}(\mathcal{D})$ is a distributive bilattice. Furthermore, every distributive bilattice is isomorphic to $\mathcal{B}(\mathcal{D})$ for some De Morgan algebra $\mathcal{D}$.

For a truth-domain $\mathcal{D}$, an element $\langle x, y\rangle$ of $\mathcal{B}(\mathcal{D})$ is interpreted as a truth value whose degree of truth is x and degree of falsity is y. For example, $\mathcal{B}(\mathbf{2})$ consists of four elements: $\langle \mathsf{t}, \mathsf{f}\rangle$ representing true – maximal degree of truth and minimal degree of falsity, $\langle \mathsf{f}, \mathsf{t}\rangle$ representing false, $\langle \mathsf{f}, \mathsf{f}\rangle$ representing lack of knowledge – minimal degree of both truth and falsity, and $\langle \mathsf{t}, \mathsf{t}\rangle$ representing an inconsistency (or disagreement) – maximal degree of both truth and falsity. It is easy to verify that $\mathcal{B}(\mathbf{2})$ is exactly Belnap logic shown in Figure 1(b). For convenience, we introduce projections π_{t} and π_{f} defined as $\pi_{\mathsf{t}}(\langle x, y\rangle) \triangleq x$ and $\pi_{\mathsf{f}}(\langle x, y\rangle) \triangleq y$.

Guided by the above intuition, we say that $\mathcal{B}(\mathcal{D})$ is an *abstract truth-domain* corresponding to a truth domain $\mathcal{D}$. Intuitively, $\langle x, y\rangle \in \mathcal{B}(\mathcal{D})$ approximates $c \in \mathcal{D}$ if x is no more true than c, and y is no more false than c. In particular, $\langle c, -c\rangle$ is the best approximation of c. Formally, this is captured by an abstraction function $\alpha_t(c) \triangleq \langle c, -c\rangle$, and a soundness relation $\rho_t \triangleq \{(a, c) \mid a \preceq \alpha_t(c)\}$.

It is easy to verify that truth operations of $\mathcal{B}(\mathcal{D})$, including negation, preserve soundness. That is, if $a_1 \preceq \alpha_t(c_1)$ and $a_2 \preceq \alpha_t(c_2)$, then $a_1 \wedge a_2 \preceq \alpha_t(c_1 \wedge c_2)$, $a_1 \vee a_2 \preceq \alpha_t(c_1 \vee c_2)$, and $\neg a_1 \preceq \alpha_t(\neg c_1)$. Furthermore, $\sqcap$ is an info-preserving widening.

3.2 Set Abstraction

We now formally define the soundness relation ρ_s between concrete ($C \to \mathcal{D}$) and abstract ($A \to \mathcal{B}(\mathcal{D})$) sets as:

$$S_\alpha \, \rho_s \, S \triangleq \forall a \in A \cdot \forall c \in \gamma_e(a) \cdot S_\alpha(a) \preceq \alpha_t(S(c)) \qquad \text{(set soundness)}$$

The soundness relation ρ_s is functional, and the corresponding abstraction function α_s follows immediately from Theorem 1:

$$\alpha_s(S)(a) \triangleq \sqcap_{c \in \gamma_e(a)} \alpha_t(S(c)) \qquad \text{(set abstraction)}$$

Note that $\alpha_s(S)(a) = \langle x, y\rangle$ means that the elements in $\gamma_e(a)$ belong to S with a truth degree of at least x, and to $\overline{S}$ with a truth degree of at least y. In particular, if S is a boolean set, then $\alpha_s(S)$ is a Belnap set; $\alpha_s(S)(a)$ is t iff $\gamma_e(a)$ is contained in S, f iff $\gamma_e(a)$ is contained in $\overline{S}$, m iff $\gamma_e(a)$ is not contained in either S or $\overline{S}$, and d iff $\gamma_e(a)$ is contained in both S and $\overline{S}$.

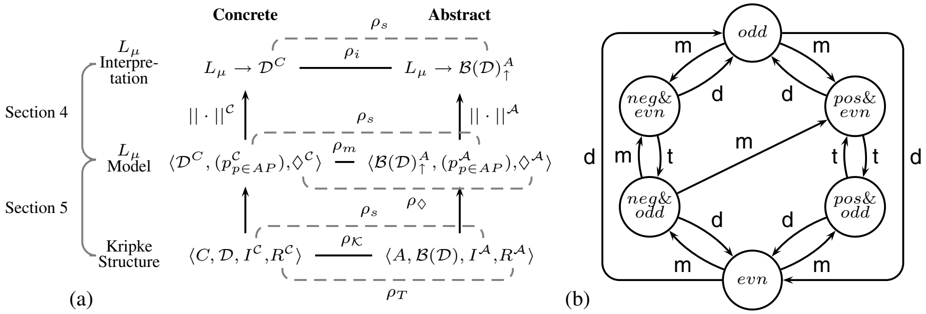


Fig. 2. (a) Abstracting L_μ: the top row summarizes soundness relations for abstracting $L\mu$ interpretations; the middle one – L_μ models, i.e., interpretions of atomic propositions and $\Diamond$ relation; the bottom one – L_μ-preserving abstractions of Kripke structures. (b) A fragment of the abstraction $\alpha_T(R_1)$, where $R_1(x) = x + 1$.

For example, an abstraction $\alpha_s(\mathit{EVEN})$ of a boolean set $\mathit{EVEN} \in \mathbf{2}^{\mathbb{Z}}$ is

$$\alpha_s(\mathit{EVEN})(a) \triangleq \begin{cases} \mathsf{t} & \text{if } \gamma_e(a) \subseteq \mathit{EVEN} \\ \mathsf{f} & \text{if } \gamma_e(a) \subseteq \mathit{ODD} \\ \mathsf{m} & \text{otherwise} \end{cases}$$

Note a difference between an abstract element evn and an abstract set $\alpha_s(\mathit{EVEN})$. The former represents a property of being an even number, and $\gamma_e(evn) = \mathit{EVEN}$ is the set of all numbers having this property. On the other hand, $\alpha_s(\mathit{EVEN})$ represents a set that contains all even and no odd numbers; hence, $\gamma_s(\alpha_s(\mathit{EVEN})) = \{\mathit{EVEN}\}$ is a singleton containing the only set satisfying these conditions.

Recall that the set operations of $\mathcal{B}(\mathcal{D})^A$ are pointwise extensions of the corresponding operations of $\mathcal{B}(\mathcal{D})$; therefore, they preserve soundness. For example, if S_α ρ_s-approximates S, then $\overline{S_\alpha}$ ρ_s-approximates $\overline{S}$, etc.

Finally, since ρ_s is functional, following the discussion in Section 2.4, we restrict the domain of abstract sets to $\preceq$-monotone functions, i.e., to $\mathcal{B}(\mathcal{D})^A_\uparrow$. Note that abstract set operations preserve $\preceq$-monotonicity and do not interfere with this restriction. This gives us with a abstract domain for sets that (a) preserves all set operations and (b) has an info-preserving widening. We use elements of this abstract domain as basic blocks for designing L_μ-preserving abstractions in the next section.

4 Abstract Interpretation for Modal μ-Calculus

In this section, we develop an abstraction of L_μ models that is sound w.r.t. satisfaction and refutation of all L_μ formulas, i.e., if an L_μ formula is satisfied (refuted) by the abstract model, it is satisfied (refuted) by the concrete one. We start by formalizing the notion of L_μ-preserving approximation in the language of AI, and then systematically extend it to the desired abstraction. The top half of the diagram in Figure 2(a) illustrates the structures and relations discussed in this section, where solid lines represent relations between structures, and dashed those between their components.

We assume that C is a collection of concrete elements, called states, and $\mathcal{D}$ is a truth domain. Recall from Section 2.3 that an interpretation of L_μ $||\cdot||$ over a set domain $\mathcal{D}^C$ maps each closed L_μ formula to a $\mathcal{D}$-set over C, where $||\varphi||(c)$ is the degree to which a formula φ is true in a state c.

Let A be an abstract domain approximating C via a soundness relation ρ_e, and $\mathcal{B}(\mathcal{D})$ be an abstract truth domain approximating $\mathcal{D}$ via a soundness relation ρ_t as defined in Section 3.1. Furthermore, let $||\cdot||_\alpha$ be an interpretation of L_μ formulas as $\mathcal{B}(\mathcal{D})$-sets over A.

A natural way to extend the soundness relation ρ_e from states to L_μ interpretations is to say that $||\cdot||_\alpha$ approximates $||\cdot||$ if for every L_μ formula φ and every abstract state $a \in A$, $||\varphi||_\alpha(a)$ approximates the degree to which $||\varphi||$ is true for every concrete state c corresponding to a. We denote this soundness relation by ρ_i and formalize it using the set soundness relation ρ_s, defined in Section 3.2, as

$$||\cdot||_\alpha \;\; \rho_i \;\; ||\cdot|| \triangleq \forall \varphi \in L_\mu \cdot ||\varphi||_\alpha \;\; \rho_s \;\; ||\varphi|| \qquad (L_\mu \text{ soundness})$$

In this paper, we are only interested in the model-based interpretations of L_μ. A natural way to extend ρ_i to models is to say that a concrete model $\mathcal{C}$ is approximated by an abstract model $\mathcal{A}$ if the corresponding L_μ interpretation $||\cdot||^{\mathcal{C}}$ is approximated by $||\cdot||^{\mathcal{A}}$. Formally, we define a model soundness relation ρ_m as

$$\mathcal{A}\,\rho_m\,\mathcal{C} \triangleq ||\cdot||^{\mathcal{A}} \;\; \rho_i \;\; ||\cdot||^{\mathcal{C}} \qquad (\text{model soundness})$$

In the rest of this section, we employ the AI framework to construct an abstract model $\mathcal{A}$ that is a best ρ_m-approximation of a given concrete model $\mathcal{C}$. As discussed in Section 3, we restrict the universe of $\mathcal{A}$ to $\preceq$-monotone functions from A to $\mathcal{B}(\mathcal{D})$.

We first outline the steps involved: (1) define a soundness relation $\rho_\Diamond$ between interpretations of the $\Diamond$ operator and derive the corresponding abstraction function $\alpha_\Diamond$; (2) show that an abstract model $\mathcal{A} = (\mathcal{B}(\mathcal{D})^A_\uparrow, (p^{\mathcal{A}})_{p\in AP}, \Diamond^{\mathcal{A}})$ ρ_m-approximates a concrete model $\mathcal{C} = (\mathcal{D}^C, (p^{\mathcal{C}})_{p\in AP}, \Diamond^{\mathcal{C}})$ if for each $p \in AP$, $p^{\mathcal{A}}$ ρ_s-approximates $p^{\mathcal{C}}$, and $\Diamond^{\mathcal{A}}$ $\rho_\Diamond$-approximates $\Diamond^{\mathcal{C}}$; (3) conclude that the best approximation of $\mathcal{C}$ is given by $\alpha_m(\mathcal{C}) \triangleq (\mathcal{B}(\mathcal{D})^A_\uparrow, (\alpha_s(p^{\mathcal{C}}))_{p\in AP}, \alpha_\Diamond(\Diamond^{\mathcal{C}}))$.

Step 1. For a given L_μ-model, an interpretation of modal formulas, i.e. formulas with $\Diamond$ but no fixpoint quantifiers, is determined by the model's interpretation of the $\Diamond$ operator. Thus, we define $\rho_\Diamond$ as follows:

$$\Diamond^{\mathcal{A}} \;\; \rho_\Diamond \;\; \Diamond^{\mathcal{C}} \triangleq \forall X \in \mathcal{B}(\mathcal{D})^A_\uparrow \cdot \forall Y \in \gamma_s(X) \cdot \Diamond^{\mathcal{A}}(X) \;\; \rho_s \;\; \Diamond^{\mathcal{C}}(Y) \qquad (\Diamond\text{-soundness})$$

Following Theorem 1, its corresponding abstraction function $\alpha_\Diamond$ is defined as

$$\alpha_\Diamond(\Diamond^{\mathcal{C}})(X) \triangleq \sqcap_{Y\in\gamma_s(X)} \; \alpha_s(\Diamond^{\mathcal{C}}(Y)) \qquad (\Diamond\text{-abstraction})$$

Step 2. To show that an abstract model $\mathcal{A}$ ρ_m-approximates a concrete model $\mathcal{C}$ if each component of $\mathcal{A}$ approximates the corresponding counterpart of $\mathcal{C}$, we need to show that for any formula φ, $||\varphi||^{\mathcal{A}}$ ρ_s-approximates $||\varphi||^{\mathcal{C}}$.

Theorem 4. *Let $\mathcal{C} = (\mathcal{D}^C, (p^{\mathcal{C}})_{p\in AP}, \Diamond^{\mathcal{C}})$ be a concrete model, $\mathcal{A} = (\mathcal{B}(\mathcal{D})^A,$ $(p^{\mathcal{A}})_{p\in AP}$, $\Diamond^{\mathcal{A}})$ be an abstract model such that A approximates C via a soundness relation ρ_e. Then, $\mathcal{A}\,\rho_m\,\mathcal{C} \Leftarrow \forall p \in AP \cdot p^{\mathcal{A}}\,\rho_s\,p^{\mathcal{C}} \wedge \Diamond^{\mathcal{A}}\,\rho_\Diamond\,\Diamond^{\mathcal{C}}$.*

The theorem is proved by structural induction on φ, using Theorem 2 for cases where φ contains a fixpoint quantifier.

Step 3. Finally, we define an abstraction function α_m that maps each concrete model to its best abstract approximation:

$$\alpha_m(\mathcal{C}) \triangleq (\mathcal{B}(\mathcal{D})^A_\uparrow, (\alpha_s(p^\mathcal{C}))_{p\in AP}, \alpha_\Diamond(\Diamond^\mathcal{C})) \qquad \text{(model abstraction)}$$

For example, consider a concrete boolean model $\mathcal{C} = (\mathbf{2}^\mathbb{Z}, p^\mathcal{C}, \Diamond^\mathcal{C})$, where $p^\mathcal{C} = \mathit{EVEN}$ and $\Diamond^\mathcal{C} = \lambda S \cdot \{y \mid y+1 \in S\}$. Then, $\Diamond p$ is interpreted in $\mathcal{C}$ as $||\Diamond p||^\mathcal{C} = \Diamond^\mathcal{C}(\mathit{EVEN}) = \mathit{ODD}$, and in the abstraction of $\mathcal{C}$ as $||\Diamond p||^{\alpha_m(\mathcal{C})} = \alpha_\Diamond(\Diamond^\mathcal{C})(\alpha_s(\mathit{EVEN})) = \alpha_s(\mathit{ODD})$.

The resulting abstraction function α_m allows us to abstract L_μ models, obtaining abstractions which are both sound and precise. However, α_m depends on an interpretation of $\Diamond$ modality, which we left unspecified. We study this subject below.

5 Abstraction of Kripke Structures

In practice, the $\Diamond$ modality is often interpreted using a Kripke structure. In this section, we are interested in conditions under which a Kripke structure over an abstract statespace (i.e., an abstract Kripke structure) is a best approximation of a given concrete one. We show that the framework of AI provides an elegant and almost mechanical way to answer this question.

Approximating Kripke Structures. Below, we aim to extend the soundness relation ρ_m between models to a soundness relation $\rho_\mathcal{K}$ between Kripke structures, and derive a corresponding abstraction function $\alpha_\mathcal{K}$.

Throughout this section, we assume that $\mathcal{C} = (C, \mathcal{D}, I^\mathcal{C}, R^\mathcal{C})$ is a concrete Kripke structure over concrete states C and a truth domain $\mathcal{D}$, and $\mathcal{A} = (A, \mathcal{B}(\mathcal{D}), I^\mathcal{A}, R^\mathcal{A})$ is an abstract Kripke structure, where A is an abstract domain related to C via ρ_e, and $\mathcal{B}(\mathcal{D})$ is an abstract truth domain related to $\mathcal{D}$ via ρ_t.

The soundness relation $\rho_\mathcal{K}$ on Kripke structures is defined as a restriction of the model soundness relation ρ_m (see Figure 2(a)):

$$\mathcal{A}\,\rho_\mathcal{K}\,\mathcal{C} \triangleq \mathcal{M}(\mathcal{A})\,\rho_m\,\mathcal{M}(\mathcal{C}) \qquad \text{(Kripke soundness)}$$

By Theorem 4, $\rho_\mathcal{K}$ is decomposed over the components of the Kripke structure:

$$\mathcal{A}\,\rho_\mathcal{K}\,\mathcal{C} \Leftarrow (\forall p \in AP \cdot I^\mathcal{A}(p)\,\rho_s\,I^\mathcal{C}(p)) \wedge R^\mathcal{A}\,\rho_T\,R^\mathcal{C}$$

where the relation ρ_T between transition functions is defined as:

$$R^\mathcal{A}\,\rho_T\,R^\mathcal{C} \triangleq \mathit{pre}[R^\mathcal{A}]\,\rho_\Diamond\,\mathit{pre}[R^\mathcal{C}] \qquad \text{(transition soundness)}$$

The abstraction function α_s corresponding to ρ_s has already been defined in Section 3.2. Thus, the only missing ingredient for defining $\alpha_\mathcal{K}$ is the transition abstraction α_T. Unfortunately, the soundness relation ρ_T is not functional; making Theorem 1 not applicable. However, we show below that ρ_T can be easily made functional. We begin by introducing an intersection operator isct: $\mathrm{isct}(X)(S) \triangleq \vee_t(X \cap S)(t)$ which allows us to express the pre-image of a transition function R as $\mathit{pre}[R](Q) = \lambda s \cdot \mathrm{isct}(R(s))(Q)$. We then define a functional soundness relation ρ_isct (see Figure 3(a)):

$$\mathrm{isct}(X)\,\rho_\mathrm{isct}\,\mathrm{isct}(Y) \triangleq \forall S \in \mathcal{B}(\mathcal{D})^A_\uparrow \cdot \forall Q \in \gamma_s(S) \cdot \mathrm{isct}(X)(S)\ \rho_t\ \mathrm{isct}(Y)(Q)$$

Noticing that $\mathrm{isct}(X)$ is determined by a set X, we extend ρ_isct to a soundness relation $\rho_\cap$ between sets (see Figure 3(b)):

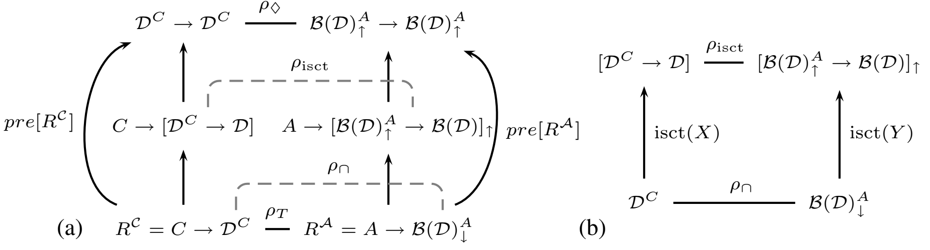


Fig. 3. (a) Soundness relations between $\Diamond$ modality and transition function; (b) Detail of (a): relations ρ_{isct} and $\rho_\cap$

$$X \,\rho_\cap\, Y \triangleq \text{isct}(X)\,\rho_{\text{isct}}\,\text{isct}(Y) \qquad \text{(successor soundness)}$$

Finally, ρ_T is made functional:

$$\begin{aligned} R^\mathcal{A}\,\rho_T\,R^\mathcal{C} &\Leftrightarrow \mathit{pre}[R^\mathcal{A}]\,\rho_\Diamond\,\mathit{pre}[R^\mathcal{C}] \\ &\Leftrightarrow \forall a \in A \cdot \forall c \in \gamma_e(s) \cdot \text{isct}(R^\mathcal{A}(a))\,\rho_{\text{isct}}\,\text{isct}(R^\mathcal{C}(c)) \\ &\Leftrightarrow \forall a \in A \cdot \forall c \in \gamma_e(s) \cdot R^\mathcal{A}(a)\,\rho_\cap\,R^\mathcal{C}(c) \end{aligned}$$

However, $\rho_\cap$ is still not functional! Thus, before applying Theorem 1 to construct α_T, we need to construct the abstraction function $\alpha_\cap$ directly, i.e., without using Theorem 1. We do so below.

Abstraction of Intersection. Intuitively, the ideal abstraction $\alpha_\cap$ is such that the diagram in Figure 3(b) commutes. That is, $\alpha_\cap(X) = Y$ implies that $\alpha_{\text{isct}}(\text{isct}(X)) = \text{isct}(Y)$. Note that ρ_{isct} is functional, thus the definition of $\alpha_{\text{isct}}(\text{isct}(X))$ follows from Theorem 1. Following a standard technique of AI, we proceed to reorganize this definition until the emergence of conditions under which $Y \in \mathcal{B}(\mathcal{D})^A$ is the best $\rho_\cap$-abstraction of X. This derivation is simple but long, and is omitted from the paper. For details, please see full version of this paper [19]. Here, we only show the final result.

Theorem 5. *Let C and $(A, \preceq_A)$ be a concrete and an abstract domain related by ρ_e, $\mathcal{D}$ and $\mathcal{B}(\mathcal{D})$ be truth-domains related by ρ_t, and for $X \in \mathcal{D}^A$, let $\alpha_\cap$ be defined as*

$$\alpha_\cap(X)(a) \triangleq \langle \vee_{c \in \gamma_e(a)} X(c), \wedge_{c \in \tilde{\gamma}_e(a)} \neg X(c) \rangle,$$

where $\tilde{\gamma}_e(a) \triangleq \{c \in C \mid \alpha_e(c) \preceq_A a\}$ is a dual-conretization function. Then, $\alpha_{\text{isct}}(\text{isct}(X)) = \text{isct}(\alpha_\cap(X))$.

To construct α_T using Theorem 1, we need an info-preserving widening. The widening $\sqcap$ on $\mathcal{B}(D)^A$ – the pointwise extension of $\sqcap$ of $\mathcal{B}(\mathcal{D})$ – is not info-preserving in general. Instead, we restrict the abstract domain to the $\preceq$-antimonotone functions, i.e., to $\mathcal{B}(\mathcal{D})^A_\downarrow$, since (a) $\mathcal{B}(\mathcal{D})^A_\downarrow$ is informationally equivalent to $\mathcal{B}(\mathcal{D})^A$, and (b) it makes pointwise widening $\sqcap$ info-preserving. Note that $\alpha_\cap(X)$ is already $\preceq$-antimonotone.

Abstraction of Transition Functions. Once the abstraction $\alpha_\cap$ is defined, the abstraction of transition functions α_T follows from Theorem 1:

$$\alpha_T(R^\mathcal{C})(a) \triangleq \sqcap_{c \in \gamma_e(a)} \alpha_\cap(R^\mathcal{C}(c)) \qquad \text{(transition abstraction)}$$

By expanding $\alpha_\cap$, α_T can be alternatively expressed as:

$$\pi_{\mathsf{t}}(\alpha_T(R^{\mathcal{C}})(a)(b)) = \wedge_{c \in \gamma_e(a)} \, pre[R^{\mathcal{C}}](\gamma_e(b))(c)$$
$$\pi_{\mathsf{f}}(\alpha_T(R^{\mathcal{C}})(a)(b)) = \wedge_{c \in \gamma_e(a)} \, \tilde{pre}[R^{\mathcal{C}}](\neg\tilde{\gamma}_e(b))(c)$$

That is, if $R^{\mathcal{A}} = \alpha_T(R^{\mathcal{C}})$, then a transition $R^{\mathcal{A}}(a)(b)$ between abstract states a and b is as true as the least degree with which all concrete states in $\gamma_e(a)$ have a successor in $\gamma_e(b)$, and as false as the least degree with which all successors of states in $\gamma_e(a)$ are *not* in $\tilde{\gamma}_e(b)$.

Note that so far, we have made no assumptions on the concrete transition function. However, if the concrete transition function $R^{\mathcal{C}}$ is boolean, then $R^{\mathcal{A}} = \alpha_T(R^{\mathcal{C}})$ is $\mathcal{B}(\mathbf{2})$-valued and satisfies:

$$R^{\mathcal{A}}(a)(b) = \langle \gamma_e(a) \subseteq pre[R^{\mathcal{C}}](\gamma_e(b)), \gamma_e(a) \subseteq \tilde{pre}[R^{\mathcal{C}}](\neg\tilde{\gamma}_e(b)) \rangle$$

For example, let $R_1(x) \triangleq x + 1$. A fragment of its abstraction $\alpha_T(R_1)$ is shown in Figure 2(b), where pos, neg and int are removed for clarity. For any even x, $x + 1$ is *definitely* odd, but it *maybe* positive or negative. Thus, the transition from evn to odd is d, and transitions to $pos\&odd$ and to $neg\&odd$ are m. Note that the pre-image of $\alpha_T(R_1)$ approximates the pre-image of R_1, e.g., $pre[\alpha_T(R_1)](\alpha_s(\mathit{EVEN})) = \alpha_s(\mathit{ODD})$.

Finally, the best abstract Kripke structure $\alpha_{\mathcal{K}}(\mathcal{C})$ of a concrete Kripke structure $\mathcal{C} = (C, \mathcal{D}, I^{\mathcal{C}}, R^{\mathcal{C}})$ is obtained compositionally:

$$\alpha_{\mathcal{K}}(\mathcal{C}) \triangleq (A, \mathcal{B}(\mathcal{D}), \alpha_s \circ I^{\mathcal{C}}, \alpha_T(R^{\mathcal{C}})) \qquad \text{(Kripke abstraction)}$$

Thus, we were able to systematically derive rules for abstracting Kripke structures by abstract Kripke structures.

Note that the diagram in Figure 3(a) does not commute, i.e., $\alpha_\Diamond(pre[R]) \neq pre[\alpha_T(R)]$. Thus, for a given Kripke structure, its best abstraction by an abstract L_μ-model is more precise than its best abstraction by an abstract Kripke structure. For example, let R_2 be

$$R_2(x) \triangleq \begin{cases} 2x & \text{if } x \geq 5 \wedge x \in \mathit{ODD} \\ -x & \text{if } 0 \leq x < 5 \wedge x \in \mathit{ODD} \\ -2 & \text{otherwise} \end{cases}$$

and $X \triangleq (\mathit{POS} \cap \mathit{EVEN}) \cup (\mathit{NEG} \cap \mathit{ODD})$. Then, $\alpha_\Diamond(pre[R_2])(\alpha_s(X))(pos\&odd) = \alpha_s(\mathit{POS} \cap \mathit{ODD})(pos\&odd) = \mathsf{t}$, but $pre[\alpha_T(R_2)](\alpha_s(X))(pos\&odd) = \mathsf{m}$. This shows that transition systems are not necessarily the best abstract domain for L_μ-preserving abstractions.

6 Application: Abstraction of Classical Kripke Structures

In this section, we look at boolean Kripke structures and compare our abstraction to that of Dams et al. [8], which provides an alternative way of computing the best L_μ-preserving abstraction of Kripke structures.

We begin by addressing minor differences between the two approaches. First, the goal of [8] is to preserve satisfaction of positive L_μ, i.e., a fragment of L_μ with negation

restricted to atomic propositions. Second, Kripke structures are abstracted by Mixed Transition Systems (MixTSs). Essentially, a MixTS is a Kripke structure with two separate transition relations, R^C and R^F, called *constrained* and *free*, respectively. The interpretation of L_μ over MixTSs is the same as its interpretation over Kripke structures, with the exception that $\Diamond$ is interpreted as $pre[R^C]$ and $\Box$ – as $\tilde{pre}[R^F]$.

Note that positive L_μ is as expressive as full L_μ: for every L_μ formula φ there exists an equivalent positive formula $NNF(\varphi)$, its negation normal form. Thus, an abstraction that preserves positive L_μ easily extends to full L_μ. Furthermore, the next theorem shows that MixTSs are equivalent to $\mathcal{B}(\mathbf{2})$-valued Kripke structures.

Theorem 6. *Let $\mathcal{T}$ be a MixTS with statespace A and transition functions R^C and R^F, and $\mathcal{K}$ be a $\mathcal{B}(\mathbf{2})$-valued Kripke structure with the same statespace, and a transition function $R^{\mathcal{K}}$ such that $R^{\mathcal{K}}(a)(b) = \langle R^C(a)(b), \neg R^F(a)(b)\rangle$. Then, for any L_μ formula φ, $||\varphi||^{\mathcal{K}} = \langle ||NNF(\varphi)||^{\mathcal{T}}, ||NNF(\neg\varphi)||^{\mathcal{T}}\rangle$.*

Thus, in the case of boolean Kripke structures, the abstraction developed in this paper is equivalent to that of [8]: same structures are used as an abstract domain, and exactly the same L_μ formulas are preserved. However, unlike the approach taken in [8], our work systematically derives both the abstraction and the notion of abstract Kripke structures from L_μ-preservation and the soundness relation ρ_s between concrete and abstract sets.

It is interesting to note that although the two abstractions are equivalent w.r.t satisfaction of L_μ, they are not identical. For completeness, Dams et al. show that the most precise MixTS abstracting a Kripke structure satisfies the following conditions:

$$R^C(a,b) \Leftrightarrow b \in \{\sqcap_{y\in Y}\alpha_e(y) \mid Y \in \min\{Y' \mid R^{\forall\exists}(\gamma_e(a), Y')\}\}$$
$$R^F(a,b) \Leftrightarrow b \in \{\sqcap_{y\in Y}\alpha_e(y) \mid Y \in \min\{Y' \mid R^{\exists\exists}(\gamma_e(a), Y')\}\}$$

where $R^{\forall\exists}(S,T) \triangleq \forall s \in S \cdot \exists t \in T \cdot R(s)(t)$ and $R^{\exists\exists}(S,T) \triangleq \exists s \in S \cdot \exists t \in T \cdot R(s)(t)$. It is different from our abstraction α_T, which, when put in this notation, is:

$$\alpha_T(R)(a)(b) = \langle R^{\forall\exists}(\gamma_e(a), \gamma_e(b)), \neg R^{\exists\exists}(\gamma_e(a), \tilde{\gamma}_e(b))\rangle$$

We believe that our characterization is simpler; however, it remains to be seen whether it is also more useful in practice, e.g., if it leads to a smaller symbolic representation, or easier to construct compositionally, etc. We leave this topic for future work.

7 Related Work

Over the years, many abstraction methods have been developed for L_μ model-checking [5, 8, 12, 17, 21, 22, 24]. They concentrate on a specific model – transition systems and most of them preserve soundness (satisfaction) for fragments of L_μ: if an abstract system is an over-approximation of the concrete one, the abstraction is sound for all universal properties. Similarly, a sound abstraction for existential properties comes from under-approximation.

The first approach for sound abstraction of *full* L_μ was proposed by Larsen and Thompsen [21]. They have shown that Modal Transition Systems (MTS) can be used to combine both over- and under-approximations. However, the goal of this work is not abstraction, and it did not consider the problem of how to abstract a Kripke structure

using an MTS. The construction problem is addressed by Dams et al. [8], who independently proposed using MixTSs, a slight generalization of MTSs, as abstract models, and provided conditions for constructing an MixTS with the best precision. Although this work uses AI to describe the relationship between concrete and abstract statespaces, abstract transition systems are not derived systematically; instead, the optimal conditions are defined based on intuition, and both soundness and optimality of precision require separate proofs.

Among the attempts of using AI to systematically derive best abstractions, the work of Loiseaux et al. [22] and Schmidt [23] are the closest to ours. [22] showed how to derive a simulation-based sound abstract transition system from Galois connections within the AI framework, but their results apply only to the universal fragment of L_μ. Motivated by the study of MixTSs, [23] showed how to capture over- and under-approximations between transition systems using AI and systematically derived Dams's most precise results. However, the starting goal of this work was formalizing the over- and the under-approximations, restricting the result to the specific L_μ models, namely, transition systems. On the other hand, in our work we start from formalizing the notion of soundness of L_μ interpretations – the most general and exact goal of abstraction for L_μ (via the soundness relation ρ_i in Section 4), and then systematically derive conditions which guarantee the best precision of the abstraction. Thus, our results can be applied to different L_μ models, where abstracting transition systems is just a special case.

Another important feature of our work is the use of bilattices. The approaches of [8, 23] develop best over- and under-approximations separately, whereas our combination of AI with bilattices provides a uniform way for abstraction of both satisfaction and refutation of L_μ. Multi-valued logic has been previously combined with abstraction in the form of 3-valued transition systems (e.g. [16]). However, these results do not use the framework of AI, and, in particular, only deal with soundness and not the precision of the abstraction. Furthermore, 3-valued Kripke structures (unlike those based on Belnap logic) lack monotonicity [24]: a more refined abstract domain does not necessarily result in a more precise abstraction, and thus the most precise abstraction may not even exist.

8 Conclusion

In this paper, we have shown that abstract interpretation provides a systematic way for designing abstractions for model-checking. On one hand, our work can be seen as recreating the pioneering work of Dams et al. [8] in a systematic setting where each step in designing an abstraction and each loss of precision can be traced back to either the choice of an abstract domain, or the requirements on the abstract structure. On the other hand, our work also extends their results to non-traditional interpretations of L_μ, such as its multi-valued [4] and quantitative [10] interpretations. To the best of our knowledge, this is the first abstraction technique that can be applied to these non-classical interpretations.

In this paper, we lay the basic groundwork for designing L_μ-preserving abstractions using the framework of AI. However, our work can be easily extended in a number of directions. We discuss a few of them below.

We have shown that requiring that an abstraction of a transition system be a transition system as well, comes with a loss of precision. Thus, it may be interesting to explore how a transition system can be abstracted directly by an abstract L_μ model. Such models will require new model-checking algorithms, but will provide additional precision, and possibly be easier to construct. For example, recent work on symmetry reduction [13] argues that instead of constructing a reduced abstract model, the symmetry-reduced $\Diamond$ modality can be implemented directly by putting symmetry reduction inside the model-checking algorithm. We believe that our framework can be used to extend this approach to other, non-symmetry induced, abstract domains. Our work on a software model-checker YASM [18] is a first step in this direction.

In designing abstractions of Kripke structures, we have assumed that the domain and range of the transition function are abstracted by the same abstract domain. This need not be the case. By using different but related abstract domains, we obtain a generalization of “hyper-transition abstractions” [24, 11] to arbitrary abstract domains.

Although not shown explicitly in the paper, the pointwise extension of the bilattice narrowing operator $\sqcup$ to abstract structures provides a simple way to combine several, not necessarily best, abstractions. This allows us to study incremental construction of abstractions, such as the one in [1].

We believe that our framework provides an interesting starting point for exploring the connection between AI and model-checking, and hope to continue this line of research in the future.

References

1. T. Ball, V. Levin, and F. Xie. “Automatic Creation of Environment Models via Training”. In *TACAS’04*, volume 2988 of *LNCS*, pages 93–107, 2004.
2. T. Ball and S. Rajamani. “The SLAM Toolkit”. In *CAV’01*, volume 2102 of *LNCS*, pages 260–264, 2001.
3. N.D. Belnap. “A Useful Four-Valued Logic”. In Dunn and Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 30–56. Reidel, 1977.
4. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. “Multi-Valued Symbolic Model-Checking”. *ACM TOSEM*, 12(4):1–38, 2003.
5. Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. *ACM TOPLAS*, 16(5):1512–1542, 1994.
6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. “Bandera: Extracting Finite-state Models from Java Source Code”. In *ICSE’00*, pages 439–448, 2000.
7. P. Cousot and R. Cousot. “Abstract Interpretation Frameworks”. *Journal of Logic and Computation*, 2(4):511–547, 1992.
8. D. Dams, R. Gerth, and O. Grumberg. “Abstract Interpretation of Reactive Systems”. *ACM TOPLAS*, 2(19):253–291, 1997.
9. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
10. L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. “Model Checking Discounted Temporal Properties”. In *TACAS’04*, volume 2988 of *LNCS*, pages 77–92, 2004.
11. L. de Alfaro, P. Godefroid, and R. Jagadeesan. “Three-Valued Abstractions of Games: Uncertainty, but with Precision”. In *LICS’04*, pages 170–179, 2004.
12. E. A. Emerson and A. P. Sistla. “Symmetry and Model Checking”. *FMSD*, 9(1-2):105–131, 1996.

13. E. A. Emerson and T. Wahl. “Dynamic Symmetry Reduction”. In *TACAS’05*, volume 3440 of *LNCS*, pages 382–396, 2005.
14. M. Fitting. “Bilattices are Nice Things”. In *Conference on Self-Reference*, 2002.
15. M. L. Ginsberg. “Multivalued Logics: A Uniform Approach to Reasoning in Artificial Intelligence”. *Computational Intelligence*, 4(3):265–316, 1988.
16. P. Godefroid, M. Huth, and R. Jagadeesan. “Abstraction-based Model Checking using Modal Transition Systems”. In *CONCUR’01*, volume 2154 of *LNCS*, pages 426–440, 2001.
17. S. Graf and H. Saïdi. “Construction of Abstract State Graphs with PVS”. In *CAV’97*, volume 1254 of *LNCS*, pages 72–83, 1997.
18. A. Gurfinkel and M. Chechik. “Yasm: Model-Checking Software with Belnap Logic”. Technical Report 533, University of Toronto, April 2005.
19. A. Gurfinkel, O. Wei, and M. Chechik. “Logical Abstract Interpretation”. Technical Report 532, University of Toronto, September 2005.
20. D Kozen. “Results on the Propositional μ-calculus”. *Theoretical Computer Science*, 27:334–354, 1983.
21. K.G. Larsen and B. Thomsen. “A Modal Process Logic”. In *LICS’88*, pages 203–210, 1988.
22. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. “Property Preserving Abstractions for the Verification of Concurrent Systems”. *FMSD*, 6:1–35, 1995.
23. D. A. Schmidt. “Closed and Logical Relations for Over- and Under-Approximation of Powersets”. In *SAS’04*, volume 3148 of *LNCS*, pages 22–37, 2004.
24. S. Shoham and O. Grumberg. “Monotonic Abstraction-Refinement for CTL”. In *TACAS’04*, LNCS, pages 546–560, April 2004.
25. O. Wei, A. Gurfinkel, and M. Chechik. “Identification and Counter Abstraction for Full Virtual Symmetry”. In *CHARME’05*, volume 3725 of *LNCS*, 2005.

Totally Clairvoyant Scheduling with Relative Timing Constraints

K. Subramani*

LDCSEE, West Virginia University, Morgantown, WV
{ksmani@csee.wvu.edu}

Abstract. Traditional scheduling models assume that the execution time of a job in a periodic job-set is constant in every instance of its execution. This assumption does not hold in real-time systems wherein job execution time is known to vary. A second feature of traditional models is their lack of expressiveness, in that constraints more complex than precedence constraints (for instance, relative timing constraints) cannot be modeled. Thirdly, the schedulability of a real-time system depends upon the degree of clairvoyance afforded to the dispatcher. In this paper, we shall discuss *Totally Clairvoyant Scheduling*, as modeled within the E-T-C scheduling framework [Sub05]. We show that this instantiation of the scheduling framework captures the central issues in a real-time flow-shop scheduling problem and devise a polynomial time sequential algorithm for the same. The design of the polynomial time algorithm involves the development of a new technique, which we term *Mutable Dynamic Programming.* We expect that this technique will find applications in other areas of system design, such as Validation and Software Verification.

1 Introduction

Real-time scheduling is concerned with the scheduling of computer jobs which are part of periodic job-sets. The execution times of these jobs are known to vary, as we move from one period to the next [AB98]. The most common cause for this feature is the presence of input-dependent loops in the program; the time taken to execute the loop structure **for(i=1 to N)**, will in general be lesser when **N=10**, than when **N=1000**. A second reason for this variance is the statistical error associated with measuring execution times [LTCA89]. Consequently the traditional approach of assuming a fixed execution time for jobs [Pin95] may not be appropriate "hard" in real-time situations, where scheduling policies should hold regardless of the actual time taken to execute by each job. Traditional models suffer from a second drawback, viz., the inability to specify complex constraints such as relative timing constraints. The literature on deterministic scheduling focuses almost exclusively on ready-time, deadline and precedence constraints [GLLK79]. In real-time applications though, there is often the necessity to constrain jobs through relationships of the form: *Start Job*

* The research of this author was supported in part by the Air Force of Scientific Research.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 398–411, 2006.
© Springer-Verlag Berlin Heidelberg 2006

J_5 *at least* 5 *units after Job* J_2 *terminates*, *Start Job* J_5 *within* 12 *units of Job* J_2 *starting*. Such relationships cannot be captured through precedence graphs, which are by definition, Directed Acyclic Graphs, whereas systems of relative timing constraints in real-time scheduling clearly contain cycles.

An important feature of any scheduling model is the schedulability predicate, i.e., what it means for a job-set to be schedulable [Sub05]. In fact, the complexity of the scheduling problem under consideration is determined in large part by the type of guarantee that we wish to provide. In this paper, our focus is on providing a polynomial time algorithm for Totally Clairvoyant Scheduling in the presence of relative timing constraints.

The principal contributions of this paper are as follows:

(a) Modeling a flow-shop problem as an instance of Totally Clairvoyant scheduling with relative timing constraint,
(b) Developing a polynomial time algorithm for this problem, and
(c) Introducing a new algorithmic technique called *Mutable Dynamic Programming* (See Sections §4 and §6).

It is to be noted that at its heart, the Totally Clairvoyant Scheduling problem is concerned with the verification of a quantified expressions. Such quantified expressions are often found in the modeling of continuous real-time and embedded systems and hence our algorithm can be thought of as an efficient verification mechanism for these kinds of problems.

2 Statement of Problem

In this section, we detail a formal description of the problem under consideration.

2.1 Job Model

Assume an infinite time-axis divided into windows of length L, starting at time $t = 0$. These windows are called *periods* or *scheduling windows.* There is a set of non-preemptive, ordered jobs, $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ that executes in each scheduling window. The occurrences of the same job in different windows are referred to as *instances* of that job. The jobs must execute in the sequence $J_1, J_2, \ldots, J_n$.

2.2 Constraint Model

The constraints on the jobs are described by System (1):

$$\mathbf{A} \cdot [\vec{\mathbf{s}}\ \ \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}, \quad \vec{\mathbf{e}} \in \mathbf{E}, \tag{1}$$

where,

(a) $\mathbf{A}$ is an $m \times 2 \cdot n$ rational matrix, $\vec{\mathbf{b}}$ is a rational $m-$ vector, $(\mathbf{A}, \vec{\mathbf{b}})$ is called the initial constraint matrix.

(b) $\mathbf{E}$ is an axis-parallel hyper-rectangle (`aph`) which is represented as the product of n closed intervals $[l_i, u_i]$, i.e.,

$$\mathbf{E} = [l_1, u_1] \times [l_2, u_2] \times \ldots \times [l_n, u_n] \tag{2}$$

We are modeling the fact that the execution time of a task can take any value in the range $[l_i, u_i]$ during actual execution and is not a fixed constant. Observe that $\mathbf{E}$ can be represented as a polyhedral system $\mathbf{M} \cdot \vec{\mathbf{e}} \leq \vec{\mathbf{m}}$, having $2 \cdot n$ constraints and n variables.

(c) $\vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n]^T$ is the start time vector of the jobs, and

(d) $\vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n]^T \in \mathbf{E}$ is the execution time vector of the jobs. We reiterate that $\vec{\mathbf{e}}$ could be different in different windows, i.e., different task instances of the same job could have different execution times (within the specified interval $[l_i, u_i]$ for J_i) in different scheduling windows. However, in any particular period, the execution time of the job is fixed and known at the start of the period.

The jobs are non-preemptive; hence the finish time of job J_i (with start time s_i) is $s_i + e_i$. The expressive power of the scheduling framework is therefore not enhanced by introducing separate finish time variables to model constraints. The ordering on the jobs is achieved by the constraint set: $s_i + e_i \leq s_{i+1}\ \forall i = 1, 2, \ldots, n-1$; these constraints are part of the $\mathbf{A}$ matrix. In the absence of ordering, the constraints on the job system cannot be captured through a polynomial-sized linear system, unless `P=NP`, since integer variables will be required to enforce non-preemption [Hoc96, Pin95].

We only permit relative timing constraints between jobs. These constraints are of the form: $s_i + e_i \leq s_j + e_j + a$, $s_i + e_i \leq s_j + a$, $s_i \leq s_j + e_j + a$, $s_i \leq s_j + a$, where a is an arbitrary integer and express relative timing (distance) constraints between the jobs J_i and J_j. As indicated, the constraints can exist between start or finish times of the jobs. Note that these constraints are a superset of absolute constraints, i.e., constraints of the form: $s_i \leq a, s_i \geq a$ or $s_i + e_i \leq a, s_i + e_i \geq a$, where a is some positive integer.

The above constraints have also been called "standard" constraints [GPS95] in the literature. We shall be using the terms "standard constraint" and relative constraint interchangeably, for the rest of the discussion.

2.3 Query Model

In the real-time applications that we consider such as Flow-Shop (see Section §3), it is possible to calculate with sufficient accuracy the execution times of the jobs in the current period and a few periods into the future. Totally Clairvoyant Scheduling assumes knowledge of the execution time of every job in the job-set, at the start of each scheduling window; the execution time vector may be different in different windows. We wish to enforce the condition that the constraints described by System (1) are met in each scheduling window, regardless of the actual execution times of the jobs. Further, the start-time vector can depend upon the execution time vector of that window.

We are now in a position to formally state the Totally Clairvoyant schedulability query:

$$\mathbf{Q} : \forall \vec{\mathbf{e}} = [e_1, e_2, \ldots, e_n] \in \mathbf{E} \;\; \exists \vec{\mathbf{s}} = [s_1, s_2, \ldots, s_n] \;\; \mathbf{A} \cdot [\vec{\mathbf{s}} \;\; \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}} \quad ? \tag{3}$$

The focus of this paper is on the design of a polynomial time procedure to decide Query (3) (henceforth **Q**).

3 Motivation

In this section, we show that a practical real-time scheduling problem can be captured as an instance of Totally Clairvoyant scheduling, with relative timing constraints.

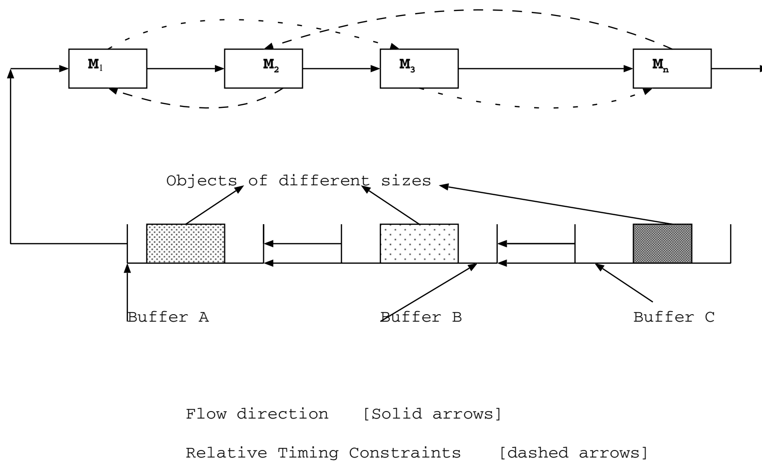


Fig. 1. Bounded-buffer Flow Shop

Figure (1) represents a bounded buffer flow shop. The flow shop consists of n machines M_1 through M_n and one or more feed-buffers (or feeders). In our example these buffers are A, B and C. Objects to be tooled also called *jobs* are placed in these buffers. The timeline on which the flow shop operates is divided into equal length portions called *periods*. At the start of each period, the job in the each buffer moves to the buffer ahead of it, while the job in the first buffer (Buffer A) enters machine M_1. Within the period, the job moves sequentially from machine M_i to machine M_{i+1}, respecting the relative timing constraints (represented by the curved, broken arrows) and finally exits at machine M_n before the end of the period. Relative timing constraints are used to capture relationships such as heating and cooling requirements; for instance the requirement that the object should wait 5 units of time after exiting machine M_1, before it enters machine M_2 is represented as: $s_2 \geq s_1 + e_1 + 5$, where s_2 is the time at

which the object enters M_2 and $s_1 + e_1$ is the time at which it exits M_1. This process is repeated in every period with new objects continuously entering the flow at the last buffer. (This example is taken from [Pin95].) Let s_i denote the time at which the machine M_i begins operating on the current job and let e_i denote the time it takes to complete its operation on the job.

Observe that the operation time of machine M_i on a job, i.e., e_i is a non-decreasing function of the job size. As shown Figure (1), the buffer pipeline is populated by jobs of different sizes and hence e_i is different for different jobs.
DESIGN PROBLEM: `Given`

1. `Timing constraints between the flow shop machines,`
2. `Lower and Upper bounds on the operation time of by each machine,`

`Does there exist a valid schedule i.e., a schedule that respects the timing constraints, for any job with size` sz, `where` $sz_{li} \leq sz \leq sz_{ui}, i = 1, 2, \ldots, n$?

The flow-shop example in this section is easily modeled as a Totally Clairvoyant scheduling problem, with the machines acting as the jobs with variable execution times.

4 Related Work

In [Sub05], we introduced the `E-T-C` scheduling framework as a model to identify and represent issues in real-time scheduling. Within the framework of that model, Zero-Clairvoyant scheduling has been addressed in [Sub02] and Partially Clairvoyant Scheduling has been detailed in [GPS95, CA00]. This is the first paper on Totally Clairvoyant scheduling. We point out that the variable elimination techniques used for Partially Clairvoyant scheduling do not seem to work in case of Totally Clairvoyant scheduling for the following reason: Standard constraints are preserved under job elimination in Partially Clairvoyant scheduling, whereas they are not preserved under job elimination in Totally Clairvoyant Scheduling. This has led us to develop a novel approach for the Totally Clairvoyant scheduling problem, which we term Mutable Dynamic Programming.

Orthogonal approaches to the issues of clairvoyance and speed have been discussed extensively in [KP00]. A number of online scheduling models with and without clairvoyance are discussed in [FW98]; however their primary concern is optimizing performance metrics in the presence of multiple processors, whereas we are concerned with checking feasibility on a single processor.

5 The Complement Problem

A simple technique to test the schedulability of a Totally Clairvoyant system is as follows: Let $\vec{\mathbf{e_1}}, \vec{\mathbf{e_2}}, \ldots \vec{\mathbf{e_l}}$ be the extreme points of $\mathbf{E}$. Substitute each extreme point of $\mathbf{E}$ in the constraint system $\mathbf{A} \cdot [\vec{\mathbf{s}} \;\; \vec{\mathbf{e}}]^{\mathbf{T}} \leq \vec{\mathbf{b}}$ and declare $\mathbf{Q}$ to be true if and only if each of the resulting linear systems (in the start-time variables) is feasible; since any execution time vector $\vec{\mathbf{e'}} \in \mathbf{E}$ can be expressed as a convex

combination of the extreme points of $\mathbf{E}$. Unfortunately such a strategy takes $\Omega(2^n)$ time, since $\mathbf{E}$ has 2^n extreme points. In this section, we shall study the complement of Query (3); our insights into the complement problem will be used to develop a polynomial time algorithm in Section §6.

Let us rewrite Query (3) as:

$$\forall \vec{\mathbf{e}} \in \mathbf{E}\ \exists \vec{\mathbf{s}}\ \mathbf{G} \cdot \vec{\mathbf{s}} + \mathbf{H} \cdot \vec{\mathbf{e}} \le \vec{\mathbf{b}},\ \vec{\mathbf{s}} \ge \vec{\mathbf{0}}? \tag{4}$$

The complement of Query (4) is:

$$\exists \vec{\mathbf{e}} \in \mathbf{E}\ \ \forall \vec{\mathbf{s}}\ \ \mathbf{G} \cdot \vec{\mathbf{s}} + \mathbf{H} \cdot \vec{\mathbf{e}} \not\le \vec{\mathbf{b}},\ \vec{\mathbf{s}} \ge \vec{\mathbf{0}}? \tag{5}$$

where the notation $\mathbf{A} \cdot \vec{\mathbf{x}} \not\le \vec{\mathbf{b}}$ means that at least one of the constraints is violated. By applying Farkas' Lemma [Sch87], we know that Query (5) is true if and only if the query:

$$\exists \vec{\mathbf{y}}\ \exists \vec{\mathbf{e}} \in \mathbf{E}\ \ \vec{\mathbf{y}} \cdot \mathbf{G} \ge \vec{\mathbf{0}},\ \vec{\mathbf{y}} \cdot (\vec{\mathbf{b}} - \mathbf{H} \cdot \vec{\mathbf{e}}) < 0,\ \vec{\mathbf{y}} \ge \vec{\mathbf{0}}? \tag{6}$$

is true.

Construct the weighted directed graph $\mathcal{G} =< \mathbf{V}, \mathbf{F}, \mathbf{c} >$ as follows:

1. Corresponding to each start time variable s_i add the vertex v_i to $\mathbf{V}$
2. Corresponding to each constraint of the form $l_p : s_i(+e_i) \le s_j(+e_j) + k$, add a directed edge of the form $v_i \leadsto v_j$ having cost $c_{l_p} = (e_j) - (e_i) + k$.

$\mathcal{G}$ is called the constraint graph corresponding to the constraint system $\mathbf{A} \cdot [\vec{\mathbf{s}}\ \ \vec{\mathbf{e}}]^{\mathbf{T}} \le \vec{\mathbf{b}}$.

Remark 5.1. *In the above construction, it is possible that there exist multiple edges between the same pair of vertices; hence, technically, $\mathcal{G}$ is a constraint multi-graph.*

Definition 1. *Let $p = v_i \leadsto v_j \leadsto \dots v_q$ denote a* simple *path in $\mathcal{G}$; the* co-static cost *of p is calculated as follows:*

1. *Symbolically add up the costs on all the edges that make up the path p to get an affine function $f(p) = \vec{\mathbf{r}} \cdot \vec{\mathbf{e}} - k$, ($\vec{\mathbf{r}} = [r_1\ r_2\ \dots\ r_n]^T, \vec{\mathbf{e}} = [e_1\ e_2\ \dots\ e_n]^T$) for suitably chosen $\vec{\mathbf{r}}$ and k. Note that each r_i belongs to the set $\{0, 1, -1\}$, since on any simple path, which is not a cycle, each vertex is encountered exactly once. Consequently, an execution time variable can be encountered at most twice, and if it is encountered twice, then the two occurrences will have opposite sign and cancel each other out.*
2. *Compute a numerical value for $f(p)$ by substituting $e_i = l_i$, if $r_i \ge 0$ and $e_i = u_i$ otherwise. This computed value is called the co-static cost of p. In other words, the co-static cost of path p is $\min_{\mathbf{E}} f(p) = \min_{\mathbf{E}}(\vec{\mathbf{r}} \cdot \vec{\mathbf{e}} - k)$.*

The co-static cost of a simple cycle is calculated similarly; if the cost of a cycle C in $\mathcal{G}$ is negative, then C is called a negative co-static cycle. The only difference between a simple path and a simple cycle is that one vertex occurs twice in this cycle. But even in this case, it is easily seen that $r_i \in \{0, 1, -1\}$.

Theorem 1. *A Totally Clairvoyant Scheduling Constraint System over a system of relative constraints has a solution if and only if its constraint graph does not have a simple negative co-static cycle.*

Proof: Assume that the constraint graph has a co-static negative cycle C_1 defined by $\{v_1 \rightsquigarrow v_2 \rightsquigarrow \ldots \rightsquigarrow v_k \rightsquigarrow v_1\}$; the corresponding set of constraints in the constraint set are:

$$\begin{aligned} s_1 - s_2 &\leq f_1(e_1, e_2) \\ s_2 - s_3 &\leq f_2(e_2, e_3) \\ &\vdots \\ s_k - s_1 &\leq f_k(e_k, e_1) \end{aligned}$$

Now, assume that there exists a solution $\vec{\mathbf{s}}$ to the constraint system. Adding up the inequalities in the above system, we get $\forall \vec{\mathbf{e}} \in \mathbf{E}\ \ 0 \leq \sum_{i=1}^{k} f_i(e_i, e_{i+1})$, where the indexes are *modulo* k. But we know that C_1 is a negative co-static cycle; it follows that $\min_{\vec{\mathbf{e}} \in \mathbf{E}} \sum_{i=1}^{k} f_i(e_i, e_{i+1}) < 0$; thus, we cannot have $\forall \vec{\mathbf{e}} \in \mathbf{E}\ \ \sum_{i=1}^{k} f_i(e_i, e_{i+1}) \geq 0$, contradicting the hypothesis.

Now consider the case, where there does not exist a negative co-static cycle. Let $\mathcal{G}_{\vec{\mathbf{e}}} =< \mathbf{V}, \mathbf{F}, \mathbf{c}_{\vec{\mathbf{e}}} >$ denote the constraint graph that results from substituting $\vec{\mathbf{e}} \in \mathbf{E}$ into the constraint system defined by System (1). It follows that for all $\vec{\mathbf{e}} \in \mathbf{E}$, $\mathcal{G}_{\vec{\mathbf{e}}}$ does not have any negative cost cycles. Hence for each $\vec{\mathbf{e}} \in \mathbf{E}$, the corresponding constraint system in the start-time variables has a solution (the vector of shortest path distances, see [CLR92]). In other words, the schedulability query $\mathbf{Q}$ is true. □

Our efforts in the next section, will be directed towards detecting the existence of negative co-static cycles in the constraint graph corresponding to a Totally Clairvoyant Scheduling Constraint system; this problem is henceforth called $\mathbf{P_1}$.

6 Mutable Dynamic Programming

In this section, we propose an algorithm for $\mathbf{P_1}$, based on Mutable Dynamic Programming. The key idea is to find the path of least co-static cost (shortest path) from each vertex $v_i \in \mathbf{V}$ to itself. By Theorem (1), we know that the constraint system is infeasible if and only if at least one of these paths has negative co-static cost.

We motivate the development of our algorithm by classifying the edges that exist between vertices in the initial constraint graph. An edge $v_i \rightsquigarrow v_j$ representing a constraint between jobs J_i and J_j must be one of the following types:

1. Type I edge: The weight of the edge does not depend upon either e_i or e_j, i.e., the corresponding constraint is expressed using only the start times of J_i and J_j. For instance, the edge corresponding to the constraint $s_i + 4 \leq s_j$ is a Type I edge.

2. Type II edge: The weight of the edge depends upon both e_i and e_j, i.e., the corresponding constraint is expressed using only the finish times of J_i and J_j. For instance, the edge corresponding to the constraint $s_i+e_i+8 \leq s_j+e_j$ is a Type II edge.
3. Type III edge: The weight of the edge depends upon e_i, but not on e_j, i.e., the corresponding constraint is expressed using the finish time of J_i and the start time of J_j. For instance, the edge corresponding to the constraint $s_i + e_i + 25 \leq s_j$ is a Type III edge.
4. Type IV edge: The weight of the edge depends upon e_j, but not on e_i, i.e., the corresponding constraint is expressed using the start time of J_i and the finish time of J_j. For instance, the edge corresponding to the constraint $s_i + 13 \leq s_j + e_j$ is a Type IV edge.

Lemma 1. *There exists at most one non-redundant $v_i \leadsto v_j$ edge of Type II.*

Proof: Without loss of generality, we assume that $i < j$, i.e., job J_i occurs in the sequence before job J_j. For the sake of contradiction, let us suppose that there exist 2 non-redundant Type II $v_i \leadsto v_j$ edges; we denote the corresponding 2 constraints as $l_1 : s_i + e_i + k_1 \leq s_j + e_j$ and $l_2 : s_i + e_i + k_2 \leq s_j + e_j$; note that they can be written as: $l_1 : s_i - s_j \leq e_j - e_i - k_1$ and $l_2 : s_i - s_j \leq e_j - e_i - k_2$. Let us say that $k_1 \geq k_2$, so that $-k_1 \leq -k_2$.

We now show that l_2 can be eliminated from the constraint set without affecting its feasibility. Note that for any fixed values of e_i and e_j, l_1 dominates l_2 in the following sense: If l_1 is satisfied, then l_2 is also satisfied. Likewise, if there is a cycle of negative co-static cost through the edge representing l_2, then there is a cycle of even lower co-static cost through the edge representing l_1. Hence l_2 can be eliminated from the constraint set, without affecting its feasibility. The case in which $i > j$ can be argued in similar fashion. □

Corollary 1. *There exists at most one non-redundant $v_i \leadsto v_j$ edge each of Types I, III and IV.*

Proof: Identical to the proof of Lemma (1). □

Corollary 2. *The number of non-redundant constraints in the initial constraint matrix which is equal to the number of non-redundant edges in the initial constraint graph is at most $O(n^2)$.*

Proof: It follows from Corollary (1) that there can exist at most 4 $i \leadsto j$ constraints and hence at most 4 $v_i \leadsto v_j$ edges between every pair of vertices v_i, v_j, $i, j = 1, 2, \ldots, n, i \neq j$. Hence the total number of edges in the initial constraint graph cannot exceed $O(8 \cdot \frac{n \cdot (n-1)}{2}) = O(n^2)$. □

We extend the taxonomy of edges discussed above to classifying paths in a straightforward way; thus a Type I path from vertex v_i to vertex v_j is a path whose cost does not depend on either e_i or e_j. Paths of Types II, III and IV are defined similarly.

Table 1. Computing the type of a path from the types of its sub-paths

$v_i \rightsquigarrow v_k$	$v_k \rightsquigarrow v_j$	$v_i \rightsquigarrow v_j$
Type I	Type I	Type I
Type I	Type II	Type IV
Type I	Type III	Type I
Type I	Type IV	Type IV
Type II	Type I	Type III
Type II	Type II	Type I (if $j = i$)
Type II	Type II	Type II (if $j \neq i$)
Type II	Type III	Type III
Type II	Type IV	Type I (if $j = i$)
Type II	Type IV	Type II (if $j \neq i$)
Type III	Type I	Type III
Type III	Type II	Type I (if $j = i$)
Type III	Type II	Type II (if $j \neq i$)
Type III	Type III	Type III
Type III	Type IV	Type I (if $j = i$)
Type III	Type IV	Type II (if $j \neq i$)
Type IV	Type I	Type I
Type IV	Type II	Type IV
Type IV	Type III	Type I
Type IV	Type IV	Type IV

Table 1 shows how to compute the type of a path, given the types of the sub-paths that constitute it.

We restrict our attention to paths and cycles of Type I; we shall see that our arguments carry over to paths and cycles of other types. As discussed above, there are at most 4 edges $v_i \rightsquigarrow v_j$, for any vertex pair (v_i, v_j). We define

$$\begin{aligned} w_{ij}(I) &= \textit{symbolic cost of the Type I edge between } v_i \textit{ and } v_j, \textit{ if such an edge exists} \\ &= \infty, \textit{ otherwise.} \end{aligned}$$

$w_{ij}(II), w_{ij}(III)$ and $w_{ij}(IV)$ are similarly defined. Note that Lemma (1) and Corollary (1) ensure that $w_{ij}(R)$ is well-defined for $R = I, II, III, IV$. By convention, $w_{ii}(R) = 0,\ i = 1, 2, \ldots, n;\ R = I, III, IV$. Note that a path of Type II from a vertex v_i to itself, is actually a Type I path!

Initialize the $n \times n \times 4$ matrix $\mathbf{W}$ as follows: $\mathbf{W}[i][j][R] = w_{ij}(R), i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, n\ R = I, II, III, IV$. Note that the entries of $\mathbf{W}$ are not necessarily numbers; for instance, if there exists a constraint of the form $s_i + e_i + 7 \leq s_j + e_j$, then $w_{ij}(II) = -e_i + e_j - 7$.

Let $p_{ij}^k(I)$ denote the path of Type I from vertex v_i to vertex v_j having the smallest co-static cost, with all intermediate vertices in the set $\{v_1, v_2, \ldots, v_k\}$, for some $k > 0$; note that $p_{ij}^0 = w_{ij}(I)$. We refer to p_{ij}^k as the *shortest Type I*

Path, from v_i to v_j, with all intermediate vertices in the set $\{v_1, v_2, \ldots, v_k\}$. Further, let $c_{ij}^k(I)$ denote the co-static cost and $d_{ij}^k(I)$ denote the corresponding symbolic cost; observe that $c_{ij}^k(I) = \min_{\mathbf{E}} d_{ij}^k(I)$ and that given $d_{ij}^k(I)$, $c_{ij}^k(I)$ can be computed in $O(n)$ time, through substitution. The quantities, $p_{ij}^k(R)$, $d_{ij}^k(R)$ and $c_{ij}^k(R)$, $R = II, III, IV$ are defined similarly.

Let us study the structure of $p_{ij}^k(I)$. We consider the following two cases.

(a) Vertex v_k is not on $p_{ij}^k(I)$ - In this case, the shortest Type I path from v_i to v_j with all the intermediate vertices in $\{v_1, v_2, \ldots, v_k\}$ is also the shortest Type I path from v_i to v_j with all the intermediate vertices in $\{v_1, v_2, \ldots, v_{k-1}\}$, i.e., $p_{ij}^k(I) = p_{ij}^{k-1}(I)$ and $d_{ij}^k(I) = d_{ij}^{k-1}(I)$.

(b) Vertex v_k is on $p_{ij}^k(I)$ - Let us assume that $j \neq i$, i.e., the path p_{ij}^k is not a cycle. From Table 1, we know that one of the following must hold (See Figure (2)):

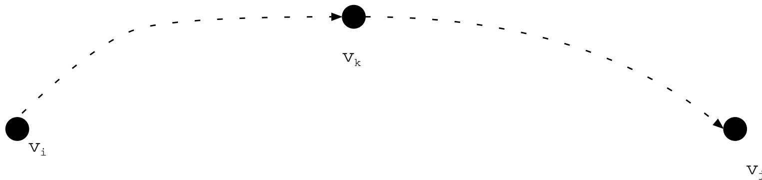


Fig. 2. Shortest path of Type I from v_i to v_j through v_k

(a) $v_i \rightsquigarrow v_k$ is of Type I and $v_k \rightsquigarrow v_j$ is of Type I - Let p_1 denote the sub-path of p_{ij}^k from v_i to v_k and let p_2 denote the sub-path of p_{ij}^k from v_k to v_j. We claim that p_1 must be the shortest Type I path from v_i to v_k with all the intermediate vertices in the set $\{v_1, v_2, \ldots, v_{k-1}\}$, i.e., $p_{ik}^{k-1}(I)$. To see this, let us assume that p_1 is not optimal and that there exists another Type I path of smaller co-static cost. Clearly this path can be combined with the existing Type I path from v_k to v_j to get a *shorter* Type I path from v_i to v_j, contradicting the optimality of $p_{ij}^k(I)$. The same argument holds for the optimality of the sub-path of p_2. This property is called the *Optimal Substructure property.* We thus have, $p_{ij}^k(I) = p_{ik}^{k-1}(I) \bigoplus p_{kj}^{k-1}(I)$ and $d_{ij}^k(I) = d_{ik}^{k-1} + d_{kj}^{k-1}$, where the $\bigoplus$ operator indicates that the combination of the 2 paths.

(b) $v_i \rightsquigarrow v_k$ is of Type I and $v_k \rightsquigarrow v_j$ is of Type III - We argue in a fashion similar to the above case to derive: $p_{ij}^k(I) = p_{ik}^{k-1}(I) \bigoplus p_{kj}^{k-1}(III)$ and $d_{ij}^k(I) = d_{ik}^{k-1}(I) + d_{kj}^{k-1}(III)$.

(c) $v_i \rightsquigarrow v_k$ is of Type IV and $v_k \rightsquigarrow v_j$ is of Type I - It follows that $p_{ij}^k(I) = p_{ik}^{k-1}(IV) \bigoplus p_{kj}^{k-1}(I)$ and $d_{ij}^k(I) = d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(I)$.

(d) $v_i \rightsquigarrow v_k$ is of Type IV and $v_k \rightsquigarrow v_j$ is of Type III - It follows that $p_{ij}^k(I) = p_{ik}^{k-1}(IV) \bigoplus p_{kj}(III)$ and $d_{ij}^k(I) = d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(III)$.

Clearly, if v_k is on $p_{ij}^k(I)$, then

$$d_{ij}^k(I) = \min_{\mathbf{E}}\{d_{ik}^{k-1}(I) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(III), \\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(III)\} \quad (7)$$

Remark 6.1. *$d_{ij}^k(I)$ represents the symbolic cost of the shortest Type I path from v_i to v_j, with all intermediate vertices in the set $\{v_1, v_2, \ldots, v_k\}$. Thus, the* $\min_{\mathbf{E}}$ *operator is used merely to select the appropriate path pairs. In particular, in the calculation of $d_{ij}^k(I)$, it does not reduce $d_{ij}^k(I)$ to a numeric value, although $c_{ij}^k(I)$ is a numeric value. It is this form of Dynamic Programming that we refer to as Mutable Dynamic Programming.*

Putting the 2 cases together, we have

$$d_{ij}^k(I) = \min_{\mathbf{E}}\{d_{ij}^{k-1}(I),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(III), \\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(III)\} \quad (8)$$

Now consider the case that the path $p_{ij}^k(I)$ is a cycle, i.e., $j = i$. From Table 1, we know that one of the following must hold:

1. $v_i \rightsquigarrow v_k$ is of Type I and $v_k \rightsquigarrow v_i$ is of Type I - This case has been handled above.
2. $v_i \rightsquigarrow v_k$ is of Type II and $v_k \rightsquigarrow v_i$ is of Type II, i.e., $d_{ii}^k(I) = d_{ik}^{k-1}(II) + d_{ki}^{k-1}(II)$.
3. $v_i \rightsquigarrow v_k$ is of Type II and $v_k \rightsquigarrow v_i$ is of Type IV, i.e., $d_{ii}^k(I) = d_{ik}^{k-1}(II) + d_{ki}^{k-1}(IV)$.
4. $v_i \rightsquigarrow v_k$ is of Type III and $v_k \rightsquigarrow v_i$ is of Type II, i.e., $d_{ii}^k(I) = d_{ik}^{k-1}(III) + d_{ki}^{k-1}(II)$.
5. $v_i \rightsquigarrow v_k$ is of Type III and $v_k \rightsquigarrow v_i$ is of Type IV, i.e., $d_{ii}^k(I) = d_{ik}^{k-1}(III) + d_{ki}^{k-1}(IV)$.

Note that the case $k = 0$, corresponds to the existence (or lack thereof) of a Type I edge from v_i to v_j. Thus, the final recurrence relation to calculate the cost of $p_{ij}^k(R),\ R = I, II, III, IV$ is:

$$\begin{aligned} d_{ij}^k(I) &= w_{ij}(I),\ if\ k = 0 \\ &= \min_{\mathbf{E}}\{d_{ik}^{k-1}(I) + d_{ki}^{k-1}(I),\ d_{ik}^{k-1}(II) + d_{ki}^{k-1}(II),\ d_{ik}^{k-1}(II) + d_{ki}^{k-1}(IV), \\ &\quad d_{ik}^{k-1}(III) + d_{ki}^{k-1}(II),\ d_{ik}^{k-1}(III) + d_{ki}^{k-1}(IV))\},\ if\ j = i \\ &= \min_{\mathbf{E}}\{d_{ij}^{k-1}(I),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(III), \\ &\quad d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(III)\},\ otherwise \end{aligned} \quad (9)$$

Using similar analyses, we derive recurrence relations for $d_{ij}^k(R), R = II, III, IV$ as follows:

$$\begin{aligned} d_{ij}^k(II) &= w_{ij}(II),\ if\ k = 0 \\ &= \min_{\mathbf{E}}\{d_{ij}^{k-1}(II),\ d_{ik}^{k-1}(II) + d_{kj}^{k-1}(II),\ d_{ik}^{k-1}(II) + d_{kj}^{k-1}(IV), \\ &\quad d_{ik}^{k-1}(III) + d_{kj}^{k-1}(II),\ d_{ik}^{k-1}(III) + d_{kj}^{k-1}(IV)\},\ otherwise \end{aligned} \tag{10}$$

$$\begin{aligned} d_{ij}^k(III) &= w_{ij}(III),\ if\ k = 0 \\ &= \min_{\mathbf{E}}\{d_{ij}^{k-1}(III),\ d_{ik}^{k-1}(II) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(II) + d_{kj}^{k-1}(III) \\ &\quad d_{ik}^{k-1}(III) + d_{kj}^{k-1}(I),\ d_{ik}^{k-1}(III) + d_{kj}^{k-1}(III)\},\ otherwise \end{aligned} \tag{11}$$

$$\begin{aligned} d_{ij}^k(IV) &= w_{ij}(IV),\ if\ k = 0 \\ &= \min_{\mathbf{E}}\{d_{ij}^{k-1}(IV),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(II),\ d_{ik}^{k-1}(I) + d_{kj}^{k-1}(IV), \\ &\quad d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(II),\ d_{ik}^{k-1}(IV) + d_{kj}^{k-1}(IV)\},\ otherwise \end{aligned} \tag{12}$$

Note that for a specific k, the values of the execution time variables, corresponding to the inner vertices of the path from v_i to v_k are fixed, by the application of the $\min_{\mathbf{E}}$ operator.

Algorithm (6.1) summarizes the above discussion on the identification of a negative co-static cycle in the constraint graph $\mathcal{G}$. We note that $\mathbf{D}_{ij}^n(I)$ represents the shortest Type I $v_i \rightsquigarrow v_j$ path with all the intermediate vertices in the set $\{v_1, v_2, \ldots, v_n\}$, i.e., it is the shortest Type I $v_i \rightsquigarrow v_j$ path. EVAL-LOOP() evaluates the co-static cost of each of the diagonal entries and declares the $\mathcal{G}$ to be co-static negative cycle free, if all entries have non-negative cost. Further, we need not consider the case $j = i$ separately, in the computations of $d_{ij}^k(R),\ R = II, III, IV$.

Remark 6.2. *We reiterate that the d_{ij}^k values are symbolic, while the c_{ij}^k values are numeric. The* $\min_E$ *operator is applied only to* select *the appropriate subpath; the d_{ij}^k cost is computed in the symbolic sense only. Once the correct subpaths have been selected, as per the principle of optimality, we can move on to the next stage. On account of the structure in the edge costs, the selection and addition procedures can be implemented in $O(n)$ time.*

6.1 Complexity

The complexity of Algorithm (6.1) is determined by Step (7 :) within the $O(n^3)$ triple loop. It is easy to see that if the symbolic costs are stored in arrays, Step (7 :) can be implemented in $O(n)$ time; it follows that Steps (1 : −10 :) can be implemented in time at most $O(n^4)$. Step (13 :) takes time at most $O(n)$ and hence Steps (11 : −18 :) take time at most $O(n^2)$.

```
Function DETECT-COSTATIC-NEGATIVE-CYCLE(G)
 1: Initialize W.
 2: Set D^0 = W.
 3: for (k = 1 to n) do
 4:    {We are determining D^k}
 5:    for (i = 1 to n) do
 6:       for (j = 1 to n) do
 7:          Compute D^k_ij(I), D^k_ij(II), D^k_ij(III), D^k_ij(IV) using the relations (9), (10),
             (11), (12).
 8:       end for
 9:    end for
10: end for
11: for (i = 1 to n) do
12:    for ( R = I to IV) do
13:       if (EVAL-LOOP(D^n_ii(R)) < 0) then
14:          return( true )
15:       end if
16:    end for
17: end for
18: return( false )
```

Algorithm 6.1. Algorithm for identifying negative co-static cycles in the constraint graph

Theorem 2. *The schedulability query for an instance of a Totally Clairvoyant scheduling problem with* n *jobs and* m *strict relative (standard) constraints can be decided in* $O(n^4)$ *time.*

7 Conclusions

In this paper, we discussed uncertainty issues in a real-time flow shop scheduling problem and designed a polynomial time algorithm for the same. The algorithm was based on a novel form of Dynamic Programming, called Mutable Dynamic Programming, which we believe may be useful in other application domains involving uncertainty and symbolic computation.

Some of the interesting open theoretical questions are as follows:

(a) What is the complexity of Totally Clairvoyant scheduling in the presence of more general constraints such as *Network Constraints* [Sub01]?,
(b) What is the complexity of finding a schedule minimizing metrics such as *Sum of Start Times* and *Sum of Completion Times*?
(c) Can we improve on the $O(n^4)$ bound derived in this paper to test Totally Clairvoyant schedulability.

We once again reiterate the importance of this technique to problems in Symbolic Model checking and Verification. Problems in these domains can be modeled as constraint verification problems and Mutable Dynamic Programming is a new procedure for these problems.

References

[AB98] Alia Atlas and A. Bestavros. Design and implementation of statistical rate monotonic scheduling in kurt linux. In *Proceedings IEEE Real-Time Systems Symposium*, December 1998.

[CA00] Seonho Choi and Ashok K. Agrawala. Dynamic dispatching of cyclic real-time tasks with relative timing constraints. *Real-Time Systems*, 19(1):5–40, 2000.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, Boston, Massachusetts, 2*nd* edition, 1992.

[FW98] Amos Fiat and Gerhard Woeginger. *Online algorithms: the state of the art*, volume 1442 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1998.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Mathematics*, 5:287–326, 1979.

[GPS95] Richard Gerber, William Pugh, and Manas Saksena. Parametric dispatching of hard real-time tasks. *IEEE Trans. Computers*, 44(3):471–479, 1995.

[Hoc96] Dorit Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, Boston, Masachusetts, 1996.

[KP00] Kalyanasundaram and Pruhs. Fault-tolerant real-time scheduling. *ALGRTHMICA: Algorithmica*, 28, 2000.

[LTCA89] S. T. Levi, S. K. Tripathi, S. D. Carson, and A. K. Agrawala. The `Maruti` Hard Real-Time Operating System. *ACM Special Interest Group on Operating Systems*, 23(3):90–106, July 1989.

[Pin95] M. Pinedo. *Scheduling: theory, algorithms, and systems.* Prentice-Hall, Englewood Cliffs, 1995.

[Sch87] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1987.

[Sub01] K. Subramani. Parametric scheduling for network constraints. In Jie Wang, editor, *Proceedings of the* 8th *Annual International Computing and Combinatorics Conference (COCOON)*, volume 2108 of *Lecture Notes in Computer Science*, pages 550–560. Springer-Verlag, August 2001.

[Sub02] K. Subramani. An analysis of zero-clairvoyant scheduling. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the* 8th *International Conference on Tools and Algorithms for the construction of Systems (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 98–112. Springer-Verlag, April 2002.

[Sub05] K. Subramani. A comprehensive framework for specifying clairvoyance, constraints and periodicty in real-time scheduling. *The Computer Journal*, 48(3):259–272, 2005.

Verification of Well-Formed Communicating Recursive State Machines*

Laura Bozzelli[1], Salvatore La Torre[2], and Adriano Peron[1]

[1] Università di Napoli Federico II , Via Cintia, 80126 - Napoli, Italy
[2] Università degli Studi di Salerno, Via S. Allende, 84081 - Baronissi, Italy

Abstract. In this paper we introduce a new (non-Turing powerful) formal model of recursive concurrent programs called well-formed communicating recursive state machines (*CRSM*). *CRSM* extend recursive state machines (*RSM*) by allowing a restricted form of concurrency: a state of a module can be refined into a finite collection of modules (working in parallel) in a potentially recursive manner. Communication is only possible between the activations of modules invoked on the same fork. We study the model checking problem of *CRSM* with respect to specifications expressed in a temporal logic that extends CaRet with a parallel operator (ConCaRet). We propose a decision algorithm that runs in time exponential in both the size of the formula and the maximum number of modules that can be invoked simultaneously. This matches the known lower bound for deciding CaRet model checking of *RSM*, and therefore, we prove that model checking *CRSM* with respect to ConCaRet specifications is Exptime-complete.

1 Introduction

Computer programs often involve the concurrent execution of multiple threads interacting with each other. Each thread can require recursive procedure calls and thus make use of a local stack. In general, combining recursion and task synchronization leads to Turing-equivalent models. Therefore, there have been essentially two approaches in the literature that address the problem of analyzing recursive concurrent programs: (i) abstraction (approximate) techniques on 'unrestricted models' (e.g. see [5, 14]), and (ii) precise techniques for 'weaker models' (with decidable reachability), obtained by imposing restrictions on the amount of parallelism and synchronization.

In the second approach, many *non* Turing-equivalent formalisms, suitable to model the control flow of recursive concurrent programs, have been proposed. One of the most powerful is constituted by *Process Rewrite Systems* (*PRS*, for short) [13], a formalism based on term rewriting, which subsumes many common infinite–state models such as Pushdown Processes, Petri Nets, and PA processes. *PRS* can be adopted as a formal model for programs with dynamic creation and (a restricted form of) synchronization of concurrent processes, and with recursive

* This research was partially supported by the MIUR grant ex-60% 2003-2004 Università degli Studi di Salerno.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 412–426, 2006.
© Springer-Verlag Berlin Heidelberg 2006

procedures (possibly with return values). *PRS* can accommodate both *parallel-call* commands and *spawn* commands for the dynamic creation of threads: in a parallel-call command, the caller thread suspends its activation waiting for the termination of all called processes, while in a spawn command, the caller thread can pursue its execution concurrently to the new activated threads. However, there is a price to pay to this expressive power: for the general framework of *PRS*, the only decidability results known in the literature concern the reachability problem between two given terms and the *reachable property* problem [13]. Model checking against standard propositional temporal logics is undecidable also for small fragments. Moreover, note that the best known upper bound for reachability of Petri nets (which represent a subclass of *PRS*) requires non primitive recursive space [9]. In [6], symbolic reachability analysis is investigated and the given algorithm can be applied only to a strict subclass of *PRS*, i.e, the *synchronization–free PRS* (the so–called PAD systems) which subsume Pushdown Processes, *synchronization–free* Petri nets, and PA processes. In [10, 11], symbolic reachability analysis of PA processes is used to allow the interprocedural data-flow analysis of programs represented by systems of *parallel flow graphs* (parallel FGS, for short), which extend classical sequential flow graphs by parallel-call commands. In [7], Dynamic Pushdown Networks (DPN) are proposed for flow analysis of multithreaded programs. DPN allows spawn commands and can encode parallel FGS. An extention of this model that captures the modelling power of PAD is also considered.

In this paper, we consider a different abstract model of concurrent programs with finite domain variables and recursive procedures, the *well-formed communicating recursive state machines* (*CRSM*). A *CRSM* is an ordered collection of finite–state machines (called *modules*) where a state can represent a call, in a potentially recursive manner, to a finite collection of modules running in parallel. A parallel call to other modules models the activation of multiple threads in a concurrent program (*fork*). When a fork happens, the execution of the current module is stopped and the control moves into the modules activated in the fork. On termination of such modules, the control return to the calling module and its execution is resumed (*join*). *CRSM* allow the communication only between module instances that are activated on the same fork and do not have ongoing procedure calls. In our model, we allow multiple entries and exits for each module, which can be used to handle finite–domain local variables and return values from procedure calls (see [1]).

Intuitively, *CRSM* correspond to the subclass of *PRS* obtained by disallowing rewrite rules which model spawn commands. Also, note that *CRSM* extend parallel FGS since they also allow (a restricted form of) synchronization and return values from (parallel) procedure calls. With respect to DPN, *CRSM* allow synchronization. Moreover, *CRSM* extend both the recursive state machines (*RSM*) [1] by allowing parallelism and the well-structured communicating (finite-state) hierarchical state machines [3] by allowing recursion. We recall that *RSM* correspond to pushdown systems, and the related model checking problem has been extensively studied in the literature [17, 4, 1]. *CRSM* are strictly more expressive

than *RSM*. In fact, it is possible to prove that *synchronization-free CRSM* correspond to a complete normal form of *Ground Tree Rewriting systems* [8] and thus the related class of languages is located in the Chomsky hierarchy strictly between the context-free and the context-sensitive languages [12].

In this paper, we address the model-checking problem of *CRSM* with respect to specifications expressed in a new logic that we call ConCaRet. ConCaRet is a linear-time temporal logic that extends CaRet [2] with a parallel operator. Recall that CaRet extends standard *LTL* allowing the specification of non-regular context-free properties that are useful to express correctness of procedures with respect to pre- and post-conditions. The model checking of *RSM* with respect to CaRet specification is known to be Exptime-complete [2].

The semantics of ConCaRet is given with respect to (infinite) computations of *CRSM*. In our model, threads can fork, thus a global state (i.e., the stack content and the current local state of each active thread) of a *CRSM* is represented as a ranked tree. Hence, a computation corresponds to a sequence of these ranked trees. As in CaRet, we consider three different notions of local successor, for any local state along a computation, and the corresponding counterparts of the usual temporal operators of *LTL* logic: the *abstract successor* captures the local computation within a module removing computation fragments corresponding to nested calls within the module; the *caller* denotes the local state (if any) corresponding to the "innermost call" that has activated the current local state; a (local) *linear successor* of a local state is the usual notion of successor within the corresponding thread. In case the current local state corresponds to a fork, its successors give the starting local states of the activated threads. With respect to the paths generated by the linear successors, we allow the standard *LTL* modalities coupled with existential and universal quantification. Note that ConCaRet has no temporal modalities for the global successor (i.e., the next global state in the run). In fact, this operator would allow us to model unrestricted synchronization among threads and thus, the model checking of ConCaRet formulas would become undecidable. In ConCaRet we also allow a parallel operator that can express properties about communicating modules such as "every time a resource p is available for a process I, then it will be eventually available for all the processes in parallel with I" (in formulas, $\Box(p \rightarrow \| \Diamond^a p)$).

We show that model-checking *CRSM* against ConCaRet is decidable. Our approach is based on automata-theoretic techniques: given a *CRSM* $\mathcal{S}$ and a formula φ, we construct a *Büchi CRSM* $\mathcal{S}_{\neg\varphi}$ (i.e., a *CRSM* equipped with generalized Büchi acceptance conditions) such that model checking $\mathcal{S}$ against φ is reduced to check the emptiness of the Büchi *CRSM* $\mathcal{S}_{\neg\varphi}$. We solve this last problem by a non-trivial reduction to the emptiness problem for a straightforward variant of classical Büchi Tree Automata. Our construction of $\mathcal{S}_{\neg\varphi}$ extends the construction given in [2] for a *RSM* and a CaRet formula. Overall, our model checking algorithm runs in time exponential in both the maximal number ρ of modules that can invoked simultaneously and the size of the formula, and thus matches the known lower bound for deciding CaRet model checking. Therefore, we prove that the model-checking problem of *CRSM* with respect to ConCaRet

specifications is EXPTIME-complete. The main difference w.r.t. *RSM* is the time complexity in the size of the model that for *RSM* is polynomial, while for *CRSM*, it is exponential in ρ.

Due to the lack of space, for the omitted details we refer the reader to [18].

2 Well-Formed Communicating Recursive State Machines

In this section we define syntax and semantics of *well-formed Communicating Recursive State Machines* (*CRSM*, for short).

Syntax. A *CRSM* is an ordered collection of finite–state machines (*FSM*) augmented with the ability of refining a state with a collection of *FSM* (working in parallel) in a potentially recursive manner.

Definition 1. *A* CRSM $\mathcal{S}$ *over a set of propositions AP is a tuple* $\langle(S_1, \ldots, S_k), start\rangle$*, where for* $1 \leq i \leq k$, $S_i = \langle\Sigma_i, \Sigma_i^s, N_i, B_i, Y_i, En_i, Ex_i, \delta_i, \eta_i\rangle$ *is a* module *and* $start \subseteq \bigcup_{i=1}^{k} N_i$ *is a set of* start nodes*. Each module* S_i *is defined as follows:*

- Σ_i *is a finite alphabet and* $\Sigma_i^s \subseteq \Sigma_i$ *is the set of* synchronization symbols*;*
- N_i *is a finite set of* nodes *and* B_i *is a finite set of* boxes *(with* $N_i \cap B_i = \emptyset$*);*
- $Y_i : B_i \to \{1, \ldots, k\}^+$ *is the* refinement function *which associates with every box a sequence of module indexes;*
- $En_i \subseteq N_i$ *(resp.,* $Ex_i \subseteq N_i$*) is a set of* entry nodes *(resp.,* exit nodes*);*
- $\delta_i : (N_i \cup Retns_i) \times \Sigma_i \to 2^{N_i \cup Calls_i}$ *is the* transition function*, where* $Calls_i = \{(b, e_1, \ldots, e_m) \mid b \in B_i,\ e_j \in En_{h_j}$ *for any* $1 \leq j \leq m$*, and* $Y_i(b) = h_1 \ldots h_m\}$ *denotes the set of* calls *and* $Retns_i = \{(b, x_1, \ldots, x_m) \mid b \in B_i$, $x_j \in Ex_{h_j}$ *for any* $1 \leq j \leq m$*, and* $Y_i(b) = h_1 \ldots h_m\}$ *denotes the set of* returns *of* S_i*; we assume w.l.o.g. that exits have no outgoing transitions, and entries have no incoming transitions, and* $En_i \cap Ex_i = \emptyset$*;*
- $\eta_i : V_i \to 2^{AP}$ *is the* labelling function*, with* $V_i = N_i \cup Calls_i \cup Retns_i$ *(*V_i *is the set of* vertices*).*

We assume that $(V_i \cup B_i) \cap (V_j \cup B_j) = \emptyset$ for $i \neq j$. Also, let $\Sigma = \bigcup_{i=1}^{i=k} \Sigma_i$, $\Sigma^s = \bigcup_{i=1}^{i=k} \Sigma_i^s$, $V = \bigcup_{i=1}^{i=k} V_i$, $Calls = \bigcup_{i=1}^{i=k} Calls_i$, $Retns = \bigcup_{i=1}^{i=k} Retns_i$, $N = \bigcup_{i=1}^{i=k} N_i$, $B = \bigcup_{i=1}^{i=k} B_i$, $En = \bigcup_{i=1}^{i=k} En_i$, and $Ex = \bigcup_{i=1}^{i=k} Ex_i$. Functions $\eta : V \to 2^{AP}$, $Y : B \to \{1, \ldots, k\}^+$, and $\delta : (N \cup Retns) \times \Sigma \to 2^{N \cup Calls}$ are defined as the natural extensions of functions η_i, Y_i and δ_i (with $1 \leq i \leq k$).

The set of states of a module is partitioned into a set of nodes and a set of boxes. Performing a transition to a box b can be interpreted as a (parallel) procedure call (*fork*) which simultaneously activates a collection of modules (the list of modules given by the refinement function Y applied to b). Since Y gives a list of module indexes, a fork can activate different instances of the same module. Note that a transition leading to a box specifies the entry node (initial state) of each activated module. All the module instances, which are simultaneously activated in a call, run in parallel, whereas the calling module instance suspends its activity waiting for the return of the (parallel) procedure call. The return

of a parallel procedure call to a box b is represented by a transition from b that specifies an exit node (exiting state) for each module copy activated by the procedural call (a synchronous return or *join* from all the activated modules).

Figure 1 depicts a simple *CRSM* consisting of two modules S_1 and S_2. Module S_1 has two entry nodes u_1 and u_2, an exit node u_4, an internal node u_3 and one box b_1 that is mapped to the parallel composition of two copies of S_2. The module S_2 has an entry node w_1, an exit node w_2, and two boxes b_2 and b_3 both mapped to one copy of S_1. The transition from node u_1 to box b_1 in S_1 is represented by a *fork* transition having u_1 as source and the entry nodes of the two copies of S_2 as targets. Similarly, the transition from box b_1 to node u_4 is represented by a *join* transition having the exit nodes of the two copies of S_2 as sources and u_4 as target.

In our model, the communication is allowed only between module instances that are activated on the same fork and are not busy in a (parallel) procedure call. As for the communicating (finite-state) hierarchical state machines [3], the form of communication we allow is synchronous and maximal in the sense that if a component (module) takes a transition labelled by a synchronization symbol σ, then each other parallel component which has σ in its synchronization alphabet must be able to take a transition labelled by σ. For instance, assuming that the symbols σ_1 and σ_2 in Figure 1 are synchronization symbols, the two copies of S_2 which refine box b_1 either both take the transition labelled by σ_1 activating a copy of S_1 with start node u_1 or both take the transition labelled by σ_2 activating a copy of S_1 with start node u_2. Transitions labelled by symbols in $\Sigma \setminus \Sigma^s$ are instead performed independently without any synchronization requirement. A *synchronization-free CRSM* is a *CRSM* in which $\Sigma^s = \emptyset$.

The *rank* of $\mathcal{S}$, written $rank(\mathcal{S})$, is the maximum of $\{|Y(b)| \mid b \in B\}$. Note that if $rank(\mathcal{S}) = 1$, then $\mathcal{S}$ is a *Recursive State Machine* (*RSM*) as defined in [1].

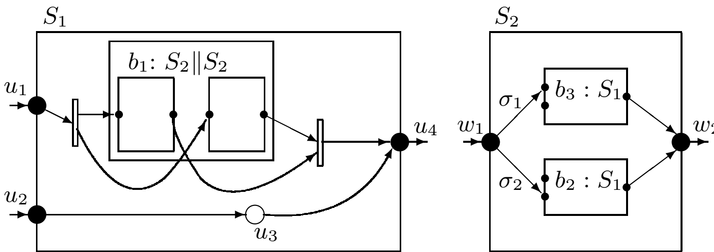


Fig. 1. A sample *CRSM*

Semantics. We give some notation first. A *tree* t is a prefix closed subset of $\mathbb{N}^*$ such that if $y \cdot i \in t$, then $y \cdot j \in t$ for any $0 \leq j < i$. The empty word ε is the *root* of t. The set of *leaves* of t is $leaves(t) = \{y \in t \mid y \cdot 0 \notin t\}$. For $y \in t$, the set of *children* of y (in t) is $children(t, y) = \{y \cdot i \in t\}$ and the set of *siblings* of y (in t) is $siblings(t, y) = \{y\} \cup \{y' \cdot i \in t \mid y = y' \cdot j \text{ for some } j \in \mathbb{N}\}$. For $y, y' \in \mathbb{N}^*$, we write $y \prec y'$ to mean that y is a proper prefix of y'.

The semantics of a $CRSM$ $\mathcal{S}$ is defined in terms of a Labelled Transition System $K_{\mathcal{S}} = \langle Q, R \rangle$. Q is the set of (global) states which correspond to activation hierarchies of instances of modules, and are represented by finite trees whose locations are labelled with vertices and boxes of the $CRSM$. Leaves correspond to active modules and a path in the tree leading to a leave y (excluding y) corresponds to the local call stack of the module instance associated with y. Formally, a state is a pair of the form (t, D) where t is a (finite) tree and $D : t \to B \cup V$ is defined as follows:

- if $y \in leaves(t)$, then $D(y) \in V$ (i.e. a vertex of $\mathcal{S}$);
- if $y \in t \setminus leaves(t)$ and $children(t, y) = \{y \cdot 0, \ldots, y \cdot m\}$, then $D(y) = b \in B$, $Y(b) = h_0 \ldots h_m$, and $D(y \cdot j) \in B_{h_j} \cup V_{h_j}$ for any $j = 0, \ldots, m$.

The *global transition relation* $R \subseteq Q \times (2^{\mathbb{N}^*} \times 2^{\mathbb{N}^*}) \times Q$ is a set of tuples of the form $\langle (t, D), (\ell, \ell'), (t', D') \rangle$ where (t, D) (resp., (t', D')) is the source (resp., target) of the transition, ℓ keeps track of the elements of t corresponding to the (instances of) modules of $\mathcal{S}$ performing the transition, and: for an *internal move* $\ell' = \ell$, for a *call*, ℓ' points to the modules activated by the (parallel) procedure call, and for a *return from a call*, ℓ' points to the reactivated caller module. Formally, $\langle (t, D), (\ell, \ell'), (t', D') \rangle \in R$ iff one of the following holds:

Single internal move: $t = t'$, there is $y \in leaves(t)$ and $\sigma \in \Sigma \setminus \Sigma^s$ such that $\ell = \ell' = \{y\}$, $D'(y) \in \delta(D(y), \sigma)$, and $D'(z) = D(z)$ for any $z \in t \setminus \{y\}$.

Synchronous internal move: $t' = t$, there are $y \in t$ with $siblings(t, y) = \{y_1, \ldots, y_m\}$, $\sigma \in \Sigma^s$, and indexes $k_1, \ldots, k_p \in \{1, \ldots, m\}$ such that the following holds: $\ell = \ell' = \{y_{k_1}, \ldots, y_{k_p}\} \subseteq leaves(t)$, $D'(z) = D(z)$ for any $z \in t \setminus \{y_{k_1}, \ldots, y_{k_p}\}$, $D'(y_{k_j}) \in \delta(D(y_{k_j}), \sigma)$ for any $1 \leq j \leq p$, and for any $j \in \{1, \ldots, m\} \setminus \{k_1, \ldots, k_p\}$, σ is *not* a synchronization symbol of the module associated with $D(y_j)$.

Module call: there is $y \in leaves(t)$ such that $D(y) = (b, e_0, \ldots, e_m) \in Call$, $t' = t \cup \{y \cdot 0, \ldots, y \cdot m\}$, $\ell = \{y\}$, $\ell' = \{y \cdot 0, \ldots, y \cdot m\}$, $D'(z) = D(z)$ for any $z \in t \setminus \{y\}$, $D'(y) = b$, and $D'(y \cdot j) = e_j$ for any $0 \leq j \leq m$.

Return from a call: there is $y \in t \setminus leaves(t)$ such that $\ell = children(t, y) = \{y \cdot 0, \ldots, y \cdot m\} \subseteq leaves(t)$, $\ell' = \{y\}$, $(D(y), D(y \cdot 0), \ldots, D(y \cdot m)) \in Retns$, $t' = t \setminus \{y \cdot 0, \ldots, y \cdot m\}$, $D'(z) = D(z)$ for any $z \in t' \setminus \{y\}$, and $D'(y) = (D(y), D(y \cdot 0), \ldots, D(y \cdot m))$.

For $v \in V$, we denote with $\langle v \rangle$ the global state $(\{\varepsilon\}, D)$ where $D(\varepsilon) = v$. A *run* of $\mathcal{S}$ is an infinite path in $K_{\mathcal{S}}$ from a state of the form $\langle v \rangle$.

2.1 Local Successors

Since we are interested in the local transformation of module instances, as in [2], we introduce different notions of local successor of module instances along a run. We fix a $CRSM$ $\mathcal{S}$ and a run of $\mathcal{S}$ $\pi = q_0 \xrightarrow{(\ell_0, \ell'_0)} q_1 \xrightarrow{(\ell_1, \ell'_1)} q_2 \ldots$ with $q_i = (t_i, D_i)$ for any i. We denote by Q_π the set $\{(i, y) \mid y \in leaves(t_i)\}$. An element (i, y) of Q_π, called *local state* of π, represents an instance of a module that at state q_i is

active and in the vertex $D_i(y)$. Note that the set $\{(i, y') \mid y' \prec y\}$ represents the (local) stack of this instance. Now, we define two notions of *next* local state (w.r.t. a run).

For $(i, y) \in Q_\pi$, $next^\ell_\pi(i, y)$ gives the set of module instances, called *linear successors* of (i, y), that are obtained by the first transition affecting (i, y). Note that such a transition may occur at $j \geq i$ or may not occur at all. In the former case $next^\ell_\pi(i, y)$ is a singleton unless $D_i(y) \in Calls$, and then the linear successors correspond to the entry nodes of the called modules. Formally, if $\{m \geq i \mid y \in \ell_m\} = \emptyset$ then $next^\ell_\pi(i, y) = \emptyset$, otherwise $next^\ell_\pi(i, y)$ is given by:

$$\{(h + 1, y') \mid h = min\{m \geq i \mid y \in \ell_m\}, y' \in \ell'_h, \text{ and either } y' \preceq y or y \preceq y'\}.$$

Note that if $rank(\mathcal{S}) = 1$ (i.e. $\mathcal{S}$ is a *RSM*), then the *linear* successor corresponds to the (standard) *global* successor.

For each $(i, y) \in Q_\pi$, we also give a notion of *abstract successor*, denoted $next^a_\pi(i, y)$. If (i, y) corresponds to a call that returns, i.e., $D_i(y) \in Calls$ and there is a $j > i$ such that $y \in leaves(t_j)$ and $D_j(y) \in Retns$, then $next^a_\pi(i, y) = (h, y)$ where h is the smallest of such j (the local state (h, y) corresponds to the matching return). For internal moves, the abstract successor coincides with the (unique) linear successor, i.e., if $next^\ell_\pi(i, y) = \{(j, y)\}$, then $next^a_\pi(i, y) = (j, y)$ (note that in this case $D_i(y) \in Retns \cup N \setminus Ex$). In all the other cases, the abstract successor is not defined and we denote this with $next^a_\pi(i, y) = \bot$. The abstract successor captures the local computations inside a module A skipping over invocations of other modules called from A.

Besides linear and abstract successor, we also define a caller of a local state (i, y) as the 'innermost call' that has activated (i, y). Formally, the *caller* of (i, y) (if any), written $next^-_\pi(i, y)$, is defined as follows (notice that only local states of the form (i, ε) have no callers): if $y = y' \cdot m$ for some $m \in \mathbb{N}$, then $next^-_\pi(i, y) = (j, y')$ where j is the maximum of $\{h < i \mid y' \in t_h$ and $D_h(y') \in Calls\}$; otherwise, $next^-_\pi(i, y)$ is undefined, written $next^-_\pi(i, y) = \bot$.

The above defined notions allow us to define sequences of local moves (i.e. moves affecting local states) in a run. For $(i, y) \in Q_\pi$, the set of *linear paths* of π *starting from* (i, y) is the set of (finite or infinite) sequences of local states $r = (j_0, y_0)(j_1, y_1) \ldots$ such that $(j_0, y_0) = (i, y)$, $(j_{h+1}, y_{h+1}) \in next^\ell_\pi(j_h, y_h)$ for any h, and either r is infinite or leads to a local state (j_p, y_p) such that $next^\ell_\pi(j_p, y_p) = \emptyset$. Analogously, the notion of *abstract path* (resp. *caller path*) of π *starting from* (i, y) can be defined by using in the above definition the abstract successor (resp. caller) instead of the linear successor. Note that a caller path is always finite and uniquely determines the content of the call stack locally to the instance of the module active at (i, y).

For module instances involved in a call, i.e., corresponding to pairs (i, y) such that $y \in t_i \setminus leaves(t_i)$, we denote the local state (if any) at which the call pending at (i, y) will return by $return_\pi(i, y)$. Formally, if $y \in leaves(t_j)$ for some $j > i$, then $return_\pi(i, y) = \{(h, y)\}$ where h is the smallest of such j, otherwise $return_\pi(i, y) = \bot$. Also, we denote the local state corresponding to the call activating the module instance at (i, y) by $call_\pi(i, y)$. Formally, $call_\pi(i, y) := (h, y)$ where h is the maximum of $\{j < i \mid y \in t_j$ and $D_j(y) \in Calls\}$.

Let *Evolve* be the predicate over sets of vertices and boxes defined as follows: $Evolve(\{v_1, \ldots, v_m\})$ holds iff *either* (1) there are $\sigma \in \Sigma \setminus \Sigma^s$ and $1 \leq i \leq m$ such that $v_i \in N \cup Retns$ and $\delta(v_i, \sigma) \neq \emptyset$ (single internal move), *or* (2) there is $\sigma \in \Sigma^s$ such that the set $H = \{1 \leq i \leq m \mid v_i \in N \cup Retns$ and $\delta(v_i, \sigma) \neq \emptyset\}$ is *not* empty and for each $i \in \{1, \ldots, m\} \setminus H$, σ does not belong to the synchronization alphabet of the module associated with v_i (synchronized internal move). In the following, we focus on *maximal runs* of *CRSM*. Intuitively, a maximal run represents an infinite computation in which each set of module instances activated by the same parallel call that may evolve (independently or by a synchronous internal move) is guaranteed to make progress.
Formally, a run π is *maximal* if for all $(i, y) \in Q_\pi$, the following holds:

- if $D_i(y) \in Calls$, then $next^\ell_\pi(i, y) \neq \emptyset$ (a possible module call must occur);
- if $y \neq \varepsilon$ and $D_i(y') \in Ex$ for all $y' \in siblings(t_i, y)$, then $next^\ell_\pi(i, y) \neq \emptyset$ (i.e., if a return from a module call is possible, then it must occur);
- if $y \neq \varepsilon$, $siblings(t_i, y) = \{y_0, \ldots, y_m\}$ and for each $0 \leq j \leq m$, $next^\ell_\pi(i, y_j) = \emptyset$ if $(i, y_j) \in Q_\pi$ and $return_\pi(i, y_j) = \bot$ otherwise, then the condition $Evolve(\{D_i(y_0), \ldots, D_i(y_m)\})$ does *not* hold.

3 The Temporal Logic ConCaRet

Let *AP* be a finite set of *atomic propositions*. The logic ConCaRet over *AP* is the set of formulas inductively defined as follows:

$$\varphi ::= p \mid call \mid ret \mid int \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^b \varphi \mid \varphi\, \mathcal{U}^b\, \varphi \mid \; \|\varphi$$

where $b \in \{\exists, \forall, a, -\}$ and $p \in AP$.

A formula is interpreted over runs of an *CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start\rangle$. Let $\pi = q_0 \xrightarrow{(\ell_0, \ell'_0)} q_1 \xrightarrow{(\ell_1, \ell'_1)} \ldots$ be a run of $\mathcal{S}$ where $q_i = (t_i, D_i)$ for $i \geq 0$. The truth value of a formula w.r.t. a local state (i, y) of π is defined as follows:

- $(i, y) \models_\pi p$ iff $p \in \eta(D_i(y))$ (where $p \in AP$);
- $(i, y) \models_\pi call$ (resp. *ret*, *int*) iff $D_i(y) \in Calls$ (resp. $D_i(y) \in Retns$, $D_i(y) \in N$);
- $(i, y) \models_\pi \neg\varphi$ iff it is not the case that $(i, y) \models_\pi \varphi$;
- $(i, y) \models_\pi \varphi_1 \vee \varphi_2$ iff either $(i, y) \models_\pi \varphi_1$ or $(i, y) \models_\pi \varphi_2$;
- $(i, y) \models_\pi \bigcirc^b\varphi$ (with $b \in \{a, -\}$) iff $next^b_\pi(i, y) \neq \bot$ and $next^b_\pi(i, y) \models_\pi \varphi$;
- $(i, y) \models_\pi \bigcirc^\exists\varphi$ iff there is $(j, y') \in next^\ell_\pi(i, y)$ such that $(j, y') \models_\pi \varphi$;
- $(i, y) \models_\pi \bigcirc^\forall\varphi$ iff for all $(j, y') \in next^\ell_\pi(i, y)$, $(j, y') \models_\pi \varphi$;
- $(i, y) \models_\pi \varphi_1 \mathcal{U}^a \varphi_2$ (resp. $\varphi_1 \mathcal{U}^- \varphi_2$) iff given the *abstract* (resp. *caller*) path $(j_0, y_0)(j_1, y_1)\ldots$ starting from (i, y), there is $h \geq 0$ such that $(j_h, y_h) \models_\pi \varphi_2$ and for all $0 \leq p \leq h-1$, $(j_p, y_p) \models_\pi \varphi_1$;
- $(i, y) \models_\pi \varphi_1 \mathcal{U}^\exists \varphi_2$ (resp. $\varphi_1 \mathcal{U}^\forall \varphi_2$) iff for some linear path (resp., for all linear paths) $(j_0, y_0)(j_1, y_1)\ldots$ starting from (i, y), there is $h \geq 0$ such that $(j_h, y_h) \models_\pi \varphi_2$ and for all $0 \leq p \leq h-1$, $(j_p, y_p) \models_\pi \varphi_1$;
- $(i, y) \models_\pi \|\varphi$ *iff* for all $y' \in siblings(t_h, y) \setminus \{y\}$ with $h = min\{j \leq i \mid (j, y) \in Q_\pi$ and $next^\ell_\pi(j, y) = next^\ell_\pi(i, y)\}$, it holds that:

– if $D_h(y') \in V$, then $(h, y') \models_\pi \varphi$;
– if $D_h(y') \in B$, then $call_\pi(h, y') \models_\pi \varphi$.

We say the run π *satisfies* a formula φ, written $\pi \models \varphi$, if $(0, \varepsilon) \models_\pi \varphi$ (recall that $t_0 = \{\varepsilon\}$). Moreover, we say $\mathcal{S}$ satisfies φ, written $\mathcal{S} \models \varphi$, iff for any $u \in start$ and for any *maximal* run π of $\mathcal{S}$ starting from $\langle u \rangle$, it holds that $\pi \models \varphi$. Now, we can define the model-checking question we are interested in:

Model checking problem: *Given a* CRSM $\mathcal{S}$ *and a* ConCaRet *formula* φ*, does* $\mathcal{S} \models \varphi$*?*

For each type of local successor (forward or backward), the logic provides the corresponding versions of the usual (global) next operator $\bigcirc$ and until operator $\mathcal{U}$. For instance, formula $\bigcirc^- \varphi$ demands that the caller of the current local state satisfies φ, while $\varphi_1 \mathcal{U}^- \varphi_2$ demands that the caller path (that is always finite) from the current local state satisfies $\varphi_1 \mathcal{U} \varphi_2$. Moreover, the linear modalities are branching-time since they quantify over the possible linear successors of the current local state. Thus, we have both existential and universal linear versions of the standard modalities $\bigcirc$ and $\mathcal{U}$. Finally, the operator $\|$ is a new modality introduced to express properties of parallel modules. The formula $\| \,\phi$ holds at a local state (i, y) of a module instance I *iff*, being $h \leq i$ the time when vertex $D_i(y)$ was first entered and such that I has been idle from h to i, any module instance (different from I) in parallel with I and not busy in a module call (at time h) satisfies φ at time h, and any module instance in parallel with I and busy in a module call (at time h) satisfies φ at the call time.

Note that the semantic of the parallel operator ensures the following desirable property for the logic ConCaRet: for each pair of local states (i, y) and (j, y) such that $i < j$ and $next^\ell_\pi(j, y) = next^\ell_\pi(i, y)$ (i.e., associated with a module instance which remains idle from i to j), the set of formulas that hold at (i, y) coincides with the set of formulas that hold at (j, y).

We conclude this section illustrating some interesting properties which can be expressed in ConCaRet. In the following, as in standard *LTL*, we will use $\Diamond^b \varphi$ as an abbreviation for $true\, \mathcal{U}^b \varphi$, for $b \in \{\exists, \forall, a, -\}$. Further, for $b \in \{a, -\}$, let $\Box^b \varphi$ stand for $\neg \Diamond^b \neg \varphi$, $\Box^\forall \varphi$ stand for $\neg \Diamond^\exists \neg \varphi$, and $\Box^\exists \varphi$ stand for $\neg \Diamond^\forall \neg \varphi$.

Besides the stack inspection properties and pre/post conditions of local computations of a module as in CaRet [2], in ConCaRet, we can express pre- and post- conditions for multiple threads activated in parallel. For instance, we can require that whenever module A and module B are both activated in parallel and pre-condition p holds, then A and B need to terminate, and post-condition q is satisfied upon the synchronous return (*parallel total correctness*). Assuming that parallel-calls to the modules A and B are characterized by the proposition $p_{A,B}$, this requirement can be expressed by the formula $\Box^\forall \big[(call \wedge p \wedge p_{A,B}) \rightarrow \bigcirc^a q\big]$. Linear modalities refer to the branching-time structure of *CRSM* computations. They can be used, for instance, to express invariance properties of the kind "every time a call occurs, then each activated module has to satisfy property φ". Such a property can be written as $\Box^\forall (call \rightarrow \bigcirc^\forall \varphi)$. We can also express simple global

eventually properties of the kind "every time the computation starts from module A, then module B eventually will be activated", expressed by the formula $t_A \rightarrow \Diamond^{\exists} t_B$. However, we can express more interesting global properties such as *recurrence* requirements. For instance, formula $\Diamond^{\forall}\Box^{\forall}(call \rightarrow \bigcirc^{\forall}\neg t_A)$ asserts that module A is activated a finite number of times. Therefore, the negation of this formula requires an interesting global recurrence property: along any maximal infinite computation module A is activated infinitely many times.

The parallel modality can express alignment properties among parallel threads. For instance, formula $\Box^{\forall}\big[call \rightarrow \bigcirc^{\exists}(\Diamond^{a} \ (\phi \wedge \parallel \phi))\big]$ requires that when a parallel-call occurs, there must be an instant in the future such that the same property ϕ holds in all the parallel threads activated by the parallel-call. In particular, with such formula we could require the existence of a future time at which all threads activated by the call will be ready for a maximal synchronization on a symbol. More generally, with the parallel operator we can express reactivity properties of modules, namely the ability of a module to continuously interact with its parallel components. We can also express mutual exclusion properties: among the modules activated in a same procedure call, at most a module can access a shared resource p (in formulas, $\Box^{\forall}(p \rightarrow \parallel \neg p)$).

4 Büchi *CRSM*

In this section, we extend *CRSM* with acceptance conditions and address the emptiness problem for the resulting class of machines (i.e., the problem about the existence of an *accepting* maximal run from a *start* node). Besides standard acceptance conditions on the finite linear paths of a maximal run π, we require a synchronized acceptance condition on modules running in parallel, and a generalized Büchi acceptance condition on the infinite linear paths of π. We call this model a Büchi *CRSM* (*B-CRSM* for short). Formally, a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_f, \mathcal{P}_{sync}\rangle$ consists of a *CRSM* $\langle (S_1, \ldots, S_k), start\rangle$ together with the following acceptance conditions:

- $\mathcal{F} = \{F_1, \ldots, F_n\}$ is a family of accepting sets of vertices of $\mathcal{S}$;
- $\mathcal{P}_F$ is the set of *terminal* vertices;
- $\mathcal{P}_{sync}$ is a predicate defined over pairs (v, H) such that v is a vertex and H is a set of vertices such that $|H| \leq rank(\mathcal{S})$.

Let $\pi = q_0 \stackrel{(\ell_0,\ell'_0)}{\longrightarrow} q_1 \stackrel{(\ell_1,\ell'_1)}{\longrightarrow} q_2 \ldots$ be a run of $\mathcal{S}$ with $q_i = (t_i, D_i)$ for $i \geq 0$. For each $i \geq 0$ and $y \in t_i$, we denote by $v(i, y)$ the vertex of $\mathcal{S}$ defined as follows: if $(i, y) \in Q_\pi$ (i.e., (i, y) is a local state), then $v(i, y) := D_i(y)$; otherwise, $v(i, y) := D_h(y)$ where $(h, y) := call_\pi(i, y)$. We say that the run π is *accepting* iff the following three conditions are satisfied:

1. for any infinite linear path $r = (i_0, y_0)(i_1, y_1)\ldots$ of π and $F \in \mathcal{F}$, there are infinitely many $h \in \mathbb{N}$ such that $D_{i_h}(y_h) \in F$ (*generalized Büchi acceptance*);
2. for any local state $(i, y) \in Q_\pi$ such that $next^{\ell}_{\pi}(i, y) = \emptyset$, condition $D_i(y) \in \mathcal{P}_F$ holds (*terminal acceptance*);

3. for any $i \geq 0$ and $y \in \ell'_i$, $\mathcal{P}_{sync}(v(i+1,y), \{v(i+1,y_1), \ldots, v(i+1,y_{m_i})\})$ holds,where $\{y_1, \ldots, y_{m_i}\}$ is $siblings(t_{i+1}, y) \setminus \{y\}$ (*synchronized acceptance*)

We say the run π is *monotone* iff for all $i \geq 0$, q_{i+1} is obtained from q_i either by a module call or by an internal move. Note that in a monotone path either the tree t_{i+1} is equal to t_i (for an internal move) or it is obtained from t_i by adding some children to a leaf (for a module call).

We decide the emptiness problem for *B-CRSM* in two main steps:

1. First, we give an algorithm to decide the problem about the existence of accepting *monotone* maximal runs starting from a given vertex;
2. Then, we reduce the emptiness problem to the problem addressed in Step 1.

4.1 Deciding the Existence of Accepting Monotone Maximal Runs

We show how to decide the existence of accepting *monotone* maximal runs in *B-CRSM* by a reduction to the emptiness problem for *Invariant Büchi tree automata*. These differ from the standard formalism of Büchi tree automata [15] for a partitioning of states into *invariant* and *non-invariant* states, with the constraint that transitions from non-invariant to invariant states are forbidden. Also, the standard Büchi acceptance condition is strengthened by requiring in addition that an accepting run must have a path consisting of invariant states only.

Formally, an (alphabet free) *invariant Büchi tree automaton* is a tuple $\mathbb{U} = \langle \mathcal{D}, P, P_0, M, F, Inv \rangle$, where $\mathcal{D} \subset \mathbb{N} \setminus \{0\}$ is a finite set of branching degrees, P is the finite set of states, $P_0 \subseteq P$ is the set of initial states, $M : P \times \mathcal{D} \to 2^{P^*}$ is the transition function with $M(s,d) \in 2^{P^d}$, for all $(s,d) \in P \times \mathcal{D}$, $F \subseteq P$ is the Büchi condition, and $Inv \subseteq P$ is the invariance condition. Also, for any $s \in P \setminus Inv$ and $d \in \mathcal{D}$, we require that if s' occurs in $M(s,d)$, then $s' \in P \setminus Inv$. A *complete* $\mathcal{D}$-tree is an infinite tree $t \subseteq \mathbb{N}^*$ such that for any $y \in t$, the cardinality of $children(t,y)$ belongs to $\mathcal{D}$. A *path* of t is a maximal subset of t linearly ordered by $\prec$. A run of $\mathbb{U}$ is a pair (t,r) where t is a complete $\mathcal{D}$-tree, $r : t \to P$ is a P-labelling of t such that $r(\varepsilon) \in P_0$ and for all $y \in t$, $(r(y \cdot 0), r(y \cdot 1), \ldots, r(y \cdot d)) \in M(r(y), d+1)$, where $d + 1 = |children(t,y)|$. The run (t,r) is *accepting* iff: (1) there is a path ν of t such that for every $y \in \nu$, $r(y) \in Inv$, and (2) for any path ν of t, the set $\{y \in \nu \mid r(y) \in F\}$ is infinite.

The algorithm in [16] for checking emptiness in Büchi tree automata can be easily extended to handle also the invariance condition, thus we obtain the following.

Proposition 1. *The emptiness problem for invariant Büchi tree automata is logspace-complete for* PTIME *and can be decided in quadratic time.*

In the following, we fix a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_F, \mathcal{P}_{sync} \rangle$.

Remark 1. Apart from a preliminary step computable in linear time (in the size of $\mathcal{S}$), we can restrict ourselves to consider only accepting monotone maximal runs π of S starting from call vertices. In fact, if π starts at a non-call vertex v of a module S_h, then either π stays within S_h forever, or π enters a call v' of S_h

that never returns. In the first case, one has to check the existence of an accepting run in the *generalized* Büchi (word) automaton given by $A_h = \langle V_h, \delta_h, \mathcal{F}_h \rangle$, where $\mathcal{F}_h$ is the restriction of $\mathcal{F}$ to the set V_h. This can be done in linear time [15]. In the second case, one has to check that there is a call v' reachable from v in A_h, and then that there is an accepting monotone maximal run in $\mathcal{S}$ from v'.

Now, we construct an invariant Büchi tree automaton $\mathbb{U}$ capturing the monotone accepting maximal runs of $\mathcal{S}$ starting from calls. The idea is to model a monotone run π of $\mathcal{S}$ as an infinite tree (a run of $\mathbb{U}$) where each path corresponds to a linear path of π. There are some technical issues to be handled.

First, there can be finite linear paths. We use symbol $\top$ to capture terminal local states of π. Therefore, the subtree rooted at the node corresponding to a terminal local state is completely labelled by $\top$. Also, since we are interested in runs of $\mathcal{S}$, we need to check that there is at least one infinite linear path in π. We do this using as invariant set the set of all $\mathbb{U}$ states except the state $\top$.

Second, when a module call associated with a box b occurs, multiple module instances $I_1, \ldots, I_m$ are activated and start running. We encode these local runs (corresponding to linear paths of π) on the same path of the run of $\mathbb{U}$ by using states of the form $(b, v_1, \ldots, v_m, i_1, \ldots, i_m, j)$, where $v_1, \ldots, v_m$ are the current nodes or calls of each module, and $i_1, \ldots, i_m, j$ are finite counters used to check the fulfillment of the Büchi condition (see below). Since in monotone runs there are no returns from calls, when a module I_j (with $1 \leq j \leq m$) moves to a call vertex v (by an internal move), we can separate the linear paths starting from v from the local runs associated with all modules $I_1, \ldots, I_m$ except I_j. Therefore, in order to simulate an internal move (in the context of modules $I_1, \ldots, I_m$), $\mathbb{U}$ nondeterministically splits in $d+1$ copies for some $0 \leq d \leq m$ such that d copies correspond to those modules (among $I_1, \ldots, I_m$) which move to call vertices, and the $d+1$-th copy goes to a state s of the form $(b, v'_1, \ldots, v'_m, i'_1, \ldots, i'_m, j')$ which describes the new status of modules $I_1, \ldots, I_m$. Note that in s, we continue to keep track of those modules which are busy in a parallel call. This is necessary for locally checking the fulfillment of the synchronized acceptance condition $\mathcal{P}_{sync}$.

The Büchi condition $\mathcal{F}$ of $\mathcal{S}$ is captured with the Büchi condition of $\mathbb{U}$ along with the use of finite counters implemented in the states. For the ease of presentation, we assume that $\mathcal{F}$ consists of a single accepting set F. Then, we use states of the form (v, i) to model a call v, where the counter i is used to check that linear paths (in the simulated monotone run of $\mathcal{S}$) containing infinite occurrences of calls satisfy the Büchi condition. In particular, it has default value 0 and is set to 1 if either $v \in F$ or a vertex in F is visited in the portion of the linear path from the last call before entering v. In the second case, the needed information is kept in the counters i_h of the states of the form $(b, v_1, \ldots, v_m, i_1, \ldots, i_m, j)$. Counter i_h has default value 0 and is set to 1 if a node in F is entered in the local computation of the corresponding module. Counter $j \in \{0, \ldots, m\}$ is instead used to check that the Büchi condition of $\mathcal{S}$ is satisfied for linear paths corresponding to infinite local computations (without nested calls) of the modules refining the box b. Moreover, in order to check that a node v_h corresponds to a terminal node (i.e., a node without linear successors in the simulated monotone run of

$\mathcal{S}$), $\mathbb{U}$ can choose nondeterministically to set the corresponding counter i_h to -1. Consistently, $\mathbb{U}$ will simulate only the internal moves from vertices $v_1, \ldots, v_m$ in which the module instance associated with v_h does not evolve. Thus, we have the following lemma.

Lemma 1. *For a call v, there is an accepting monotone maximal run of $\mathcal{S}$ from $\langle v \rangle$ iff there is an accepting run in $\mathbb{U}$ starting from $(v, 0)$.*

When the Büchi condition consists of $n > 1$ accepting sets, the only changes in the above construction concern the counters: we need to check that each set is met and thus the $0-1$ counters become counters up to n and the other counter is up to $m \cdot n$. Therefore, denoting $\rho = rank(\mathcal{S})$, n_V the number of vertices of $\mathcal{S}$ and n_δ the number of transitions of $\mathcal{S}$, we have that the number of $\mathbb{U}$ states is $O(\rho \cdot n^{\rho+1} \cdot n_V^{\rho+1})$ and the number of $\mathbb{U}$ transitions is $O(\rho^2 \cdot n^{2\rho+2} \cdot n_V \cdot (n_V + n_\delta)^\rho)$. Thus, by Proposition 1, Remark 1, and Lemma 1 we obtain the following result.

Lemma 2. *The problem of checking the existence of accepting monotone maximal runs in a* B-CRSM *$\mathcal{S}$ can be decided in time $O(\rho^4 \cdot n^{4\rho+4} \cdot (n_V + n_\delta)^{2\rho+2})$.*

4.2 The Emptiness Problem for Büchi *CRSM*

In this subsection, we show that the emptiness problem for Büchi *CRSM* can be reduced to check the existence of accepting monotone maximal runs. We fix a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_F, \mathcal{P}_{sync} \rangle$. Moreover, n_V (resp., n_δ) denotes the number of vertices (resp., transitions) of $\mathcal{S}$. Also, let $\rho := rank(\mathcal{S})$.

For $F \subseteq V$, a *finite* path of $K_\mathcal{S}$ $\pi = q_0 \stackrel{(\ell_0, \ell'_0)}{\longrightarrow} q_1 \stackrel{(\ell_0, \ell'_0)}{\longrightarrow} \ldots q_n$ (with $q_i = (t_i, D_i)$ for any $0 \leq i \leq n$, and $t_0 = \{\varepsilon\}$) is *F-accepting* iff π satisfies the synchronized acceptance condition $\mathcal{P}_{sync}$ and all the linear paths of π starting from the local state $(0, \varepsilon)$ contain occurrences of local states (i, z) such that $D_i(z) \in F$. For a box $b \in B$, we say π is a *b-path* if $D_i(\varepsilon) = b$ for all $1 \leq i \leq n-1$.
We need the following preliminary result.

Lemma 3 (Generalized Reachability Problem). *Given $F \subseteq V$, the set of pairs (v, v') such that $v = (b, e_1, \ldots, e_m)$ is a call, $v' = (b, x_1, \ldots, x_m)$ is a matching return, and there is an F-accepting b-path from $\langle v \rangle$ to $\langle v' \rangle$, can be* computed *in time $O(n_V^2 \cdot 4^\rho \cdot (n_V + n_\delta)^\rho)$.*

Now, we show how to solve the emptiness problem for *B-CRSM* using the results stated by Lemmata 2 and 3. Starting from the *B-CRSM* $\mathcal{S}$ with $\mathcal{F} = \{F_1, \ldots, F_n\}$, we construct a new *B-CRSM* $\mathcal{S}'$ such that emptiness for $\mathcal{S}$ reduces to check the existence of accepting *monotone* maximal runs in $\mathcal{S}'$.

$\mathcal{S}' = \langle (S'_1, \ldots, S'_k), start, \mathcal{F}', \mathcal{P'}_f, \mathcal{P'}_{sync} \rangle$, with $\mathcal{F}' = \{F'_1, \ldots, F'_n\}$, is defined as follows. For $1 \leq i \leq k$, S'_i is obtained extending the set of nodes and the transition function of S_i as follows. For any call $v = (b, e_1, \ldots, e_m)$ of S_i and matching return $v' = (b, x_1, \ldots, x_m)$ such that there is a V-accepting b-path in $\mathcal{S}$ from $\langle v \rangle$ to $\langle v' \rangle$, we add two new nodes u^c_{new} and u^r_{new}, and the edge $(u^c_{new}, \perp, u^r_{new})$, where $\perp$ is a fresh non-synchronization symbol. We say u^c_{new}

(resp., u^r_{new}) is *associated* with the call v (resp., return v'). Moreover, for any edge in S_i of the form (u, σ, v) (resp., of the form (v', σ, u)) we add in S'_i the edge (u, σ, u^c_{new}) (resp., the edge (u^r_{new}, σ, u)). Also, for $1 \leq i \leq n$, if there is an F_i-accepting b-path from $\langle v \rangle$ to $\langle v' \rangle$, then we add u^r_{new} to F'_i (F'_i also contains all elements of F_i). Still, if $v' \in \mathcal{P}_f$, then we add u^r_{new} to $\mathcal{P}'_f$ ($\mathcal{P}'_f$ also contains all elements of $\mathcal{P}_f$). Note that $u^c_{new} \notin \mathcal{P}_f$. In fact, if an accepting maximal run of $\mathcal{S}'$ has a local state labelled by u^c_{new}, then the linear successor of this local state is defined and is labelled by u^r_{new}. Finally, $\mathcal{P}'_{sync}(v'_0, \{v'_1, \ldots, v'_m\})$ (with $m \leq rank(\mathcal{S})$) holds iff there are $v_0, \ldots, v_m \in V$ such that $\mathcal{P}_{sync}(v_0, \{v_1, \ldots, v_m\})$ holds and for all $0 \leq j \leq m$, either $v'_j = v_j$, or v_j is a return (resp., a call) and v'_j is a "new" node associated with it. Thus, we obtain the following result.

Lemma 4. *For any node u of $\mathcal{S}$, there is an accepting maximal run in $\mathcal{S}$ from $\langle u \rangle$ iff there is an accepting* monotone *maximal run in $\mathcal{S}'$ from $\langle u \rangle$.*

Note that the number of new nodes is bounded by $2n_V^2$, the number of new edges is bounded by $n_V \cdot n_\delta + n_V^2$, and by Lemma 3, $\mathcal{S}'$ can be constructed in time $O(|\mathcal{F}| \cdot n_V^2 \cdot 4^\rho \cdot (n_V + n_\delta)^\rho)$. Thus, by Lemmata 2 and 4 we obtain the main result of this section.

Theorem 1. *Given a* B-CRSM *$\mathcal{S}$, the problem of checking the emptiness of $\mathcal{S}$ can be decided in time $O\big((|\mathcal{F}| \cdot (n_V + n_\delta))^{O(\rho)}\big)$.*

5 Model Checking *CRSM* Against ConCaRet

In this section, we solve the model-checking problem of *CRSM* against ConCaRet using an automata-theoretic approach: for a *CRSM* $\mathcal{S}$ and a ConCaRet formula φ, we construct a *B-CRSM* $\mathcal{S}_\varphi$ which has an accepting maximal run iff $\mathcal{S}$ has a maximal run that satisfies φ. More precisely, an accepting maximal run of $\mathcal{S}_\varphi$ corresponds to a maximal run π of $\mathcal{S}$ where each local state is equipped with the information concerning the set of subformulas of φ that hold at it along π.

The construction proposed here follows and extends that given in [2] for CaRet. The extensions are due to the presence of the branching-time modalities and the parallel operator $\|$. For branching-time modalities we have to ensure that the existential (resp. universal) next requirements are met in some (in each) linear successor of the current local state. This is captured locally in the transitions of $\mathcal{S}_\varphi$. Parallel formulas are handled instead by the synchronization predicate.

The generalized Büchi condition is used to guarantee the fulfillment of liveness requirements φ_2 in until formulas of the form $\varphi_1 \, \mathcal{U}^b \, \varphi_2$ where $b \in \{a, \exists, \forall\}$ (caller-until formulas do not require such condition since a caller-path is always finite). For existential until formulas φ', when φ' is asserted at a local state (i, y), we have to ensure that φ' is satisfied in at least one of the linear paths from (i, y). In order to achieve this and ensure the acceptance of all infinite linear paths from (i, y) we use a fresh atomic proposition $\tau_{\varphi'}$.

For every vertex/edge in $\mathcal{S}$, we have $2^{O(|\varphi| \cdot rank(\mathcal{S}))}$ vertices/edges in $\mathcal{S}_\varphi$. Also, the number of accepting sets in the generalized Büchi conditions is at most $O(|\varphi|)$ and $rank(\mathcal{S}_\varphi) = rank(\mathcal{S})$. Since there is a maximal run of $\mathcal{S}$ satisfying formula φ

iff there is an accepting maximal run of $\mathcal{S}_\varphi$, model checking $\mathcal{S}$ against φ is reduced to check emptiness for the Büchi *CRSM* $\mathcal{S}_{\neg\varphi}$. For $rank(\mathcal{S}) = 1$, the considered problem coincides with the model checking problem of *RSM* against CaRet that is Exptime-complete (even for a fixed *RSM*). Therefore, by Theorem 1, we obtain the following result.

Theorem 2. *For a* CRSM $\mathcal{S}$ *and a formula* φ *of* ConCaRet*, the model checking problem for* $\mathcal{S}$ *against* φ *can be decided in time exponential in* $|\varphi| \cdot (rank(\mathcal{S}))^2$*. The problem is* Exptime*-complete* (*even when the* CRSM *is fixed*).

References

1. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *To appear in ACM Transactions on Programming Languages and Systems*, 2005.
2. R. Alur, K. Etessami, and P. Madhusudan. A Temporal Logic of Nested Calls and Returns. In *Proc. of TACAS'04*, pp. 467–481, 2004.
3. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Proc. of ICALP'99*, LNCS 1644, pp. 169–178, 1999.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. of CONCUR'97*, LNCS 1243, pp. 135–150, 1997.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proc. of POPL'03*, pp. 62–73, 2003.
6. A. Bouajjani and T. Touili. Reachability Analysis of Process Rewrite Systems. In *Proc. of FSTTCS'03*, LNCS 2914, pp. 74–87, 2003.
7. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. In *Proc. of CONCUR'05*, LNCS 3653, pp. 473–487, 2005.
8. W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14: pp. 217–231, 1969.
9. J. Esparza, and M. Nielsen, Decidability Issues for Petri Nets.In J. Inform. Process. Cybernet., EIK 30 (1994) 3, pp. 143–160.
10. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural parallel flow graphs. In *Proc. of FoSSaCS'99*, LNCS 1578, 1999.
11. J. Esparza and A. Podelski. Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs. In *Proc. of POPL'00*, pp. 1–11, 2000.
12. C. Löding. Infinite Graphs Generated by Tree Rewriting. Doctoral thesis, RWTH Aachen, 2003.
13. R. Mayr. Process Rewrite Systems. In *Information and Computation*, Vol. 156, 2000, pp. 264–286.
14. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of TACAS'05*, LNCS 3440, pp. 93–107, , 2005.
15. W. Thomas. Automata on infinite objets. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 133–191, 1990.
16. M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, 1986.
17. I. Walukiewicz. Pushdown processes: Games and Model Checking. In *Int. Conf. on Computer Aided Verification*, LNCS 1102, pages 62–74. Verlag, 1996.
18. URL: www.dia.unisa.it/professori/latorre/Papers/concaret.ps.gz

What's Decidable About Arrays?*

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma

Computer Science Department, Stanford University,
Stanford, CA 94305-9045
{arbrad, zm, sipma}@theory.stanford.edu

Abstract. Motivated by applications to program verification, we study a decision procedure for satisfiability in an expressive fragment of a theory of arrays, which is parameterized by the theories of the array elements. The decision procedure reduces satisfiability of a formula of the fragment to satisfiability of an equisatisfiable quantifier-free formula in the combined theory of equality with uninterpreted functions (EUF), Presburger arithmetic, and the element theories. This fragment allows a constrained use of universal quantification, so that one quantifier alternation is allowed, with some syntactic restrictions. It allows expressing, for example, that an assertion holds for all elements in a given index range, that two arrays are equal in a given range, or that an array is sorted. We demonstrate its expressiveness through applications to verification of sorting algorithms and parameterized systems. We also prove that satisfiability is undecidable for several natural extensions to the fragment. Finally, we describe our implementation in the πVC verifying compiler.

1 Introduction

Software verification — whether via the classical Floyd-Hoare-style proof method with some automatic invariant generation, or through automatic methods like predicate abstraction — relies on the fundamental technology of decision procedures. Therefore, the properties of software that can be proved automatically are to a large extent limited by the expressiveness of the underlying fragments of theories for which satisfiability is decidable and can be checked efficiently.

Arrays are a basic data structure of imperative programming languages. Theories for reasoning about the manipulation of arrays in programs have been studied intermittently for about as long as computer science has been a recognized field [5]. Nonetheless, the strongest predicate about arrays that appears in a decidable fragment is equality between two unbounded arrays [7]. For software verification, unbounded equality is not enough: for example, assertions such as that two subarrays are equal or that all elements of a subarray satisfy a certain property are not uncommon in normal programming tasks. We study a fragment

* This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939. The first author was additionally supported by a Sang Samuel Wang Stanford Graduate Fellowship.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 427–442, 2006.
© Springer-Verlag Berlin Heidelberg 2006

of a theory of arrays that allows expressing such properties and many others, and for which satisfiability is decidable.

Various theories of arrays have been addressed in past work. Research in satisfiability decision procedures has focused on the quantifier-free fragments of array theories, as the full theories are undecidable (see Section 5). In our discussion, we use the sorts array, elem, and index for arrays, elements, and indices, respectively. The syntax $a[i]$ represents an array read, while $a\{i \leftarrow e\}$ represents the array with position i modified to e, for array a, elem e, and index i. McCarthy proposed the main axiom of arrays, read-over-write [5]:

$$(\forall \text{ array } a)(\forall \text{ elem } e)(\forall \text{ index } i, j) \\ \begin{bmatrix} & i = j & \rightarrow & a\{i \leftarrow e\}[j] = e \\ \wedge & i \neq j & \rightarrow & a\{i \leftarrow e\}[j] = a[j] \end{bmatrix}$$

An *extensional* theory of arrays has been studied formally, most recently in [7] and [1]. The extensional theory relates equations between arrays and equations between their elements:

$$(\forall \text{ array } a, b)[(\forall \text{ index } i)\ a[i] = b[i] \ \rightarrow \ a = b]$$

In [8], a decidable quantifier-free fragment of an array theory that allows a restricted use of a *permutation* predicate is studied. Their motivation, as with our work, is that verification of software requires decision procedures for expressive assertion languages. They use their decision procedure to prove that various sorting algorithms return a permutation of their input. In the conclusion of [8], they suggest that a predicate expressing the sortedness of arrays would be useful.

The main theory of arrays that we study in this paper is motivated by practical requirements in software verification. We use Presburger arithmetic for our theory of indices, so the abstract sort index is concrete for us. Additionally, the theory is *parameterized* by the element theories used to describe the contents of arrays. Typical element theories include the theory of integers, the theory of reals, and the theory of equality.

Our satisfiability decision procedure is for a fragment, which we call the *array property* fragment, that allows a constrained use of universal quantification. We characterize the fragment in Section 2, but for now we note that the decidable fragment is capable of expressing *array equality*, the usual equality in an extensional theory of arrays; *bounded equality*, equality between two subarrays; and various properties, like sortedness, of (sub)arrays.

The satisfiability procedure reduces satisfiability of a formula of the array property fragment to satisfiability of a quantifier-free formula in the combined theory of equality with uninterpreted functions (EUF), Presburger arithmetic, and the element theories. The original formula is equisatisfiable to the reduced formula. For satisfiability, handling existential quantification is immediate. Universally quantified assertions are converted to finite conjunctions by instantiating the quantified index variables over a finite set of index terms. The main insight of the satisfiability decision procedure, then, is that for a given formula in the fragment, there is a finite set of index terms such that instantiating universally

quantified index variables from only this set is sufficient for completeness (and, trivially, soundness).

After presenting and analyzing this decision procedure, we study a theory of *maps*, which are like arrays except that indices are uninterpreted. Therefore, the decidable fragment of the theory is less powerful for reasoning about arrays; however, it is more expressive than, for example, the quantifier-free fragment of the extensional theory presented in [7]. In particular, it is expressive enough to reason about hashtables.

The paper is organized as follows. Section 2 defines the theory and the fragment that we study. Section 3 describes the decision procedure for satisfiability of the fragment. In Section 4, we prove that the procedure is sound and complete. We also prove that when satisfiability for quantifier-free formulae of the combined theory of EUF, Presburger arithmetic, and array elements is in NP, then satisfiability for *bounded* fragments is NP-complete. In Section 5, we prove that several natural extensions to the fragment result in fragments for which satisfiability is undecidable; we identify one slightly larger fragment for which decidability remains open. Section 6 presents and analyzes a parametric theory of maps. Section 7 motivates the theories with several applications in software verification. We implemented the procedure in our verifying compiler $\pi\mathsf{VC}$; we describe our experience and results in Section 7.4.

2 An Array Theory and Fragment

We introduce the theory of arrays and the *array property* fragment for which satisfiability is decidable.

Definition 1 (Theories). The theory of arrays uses Presburger arithmetic, $T_{\mathbb{Z}}$, for array indices, and the parameter element theories $T^1_{\mathsf{elem}}, \ldots, T^m_{\mathsf{elem}}$, for $m > 0$, for its elements. The many-sorted array theory for the given element theories is called $T_{\mathsf{A}}^{\{\mathsf{elem}_k\}_k}$. We usually drop the superscript.

Recall that the signature of Presburger arithmetic is

$$\Sigma_{\mathbb{Z}} = \{0, 1, +, -, =, <\} .$$

Assume each T^k_{elem} has signature Σ^k_{elem}. T_{A} then has signature

$$\Sigma_{\mathsf{A}} = \Sigma_{\mathbb{Z}} \cup \bigcup_k \Sigma^k_{\mathsf{elem}} \cup \{\cdot[\cdot], \cdot\{\cdot \leftarrow \cdot\}\}$$

where the two new functions are *read* and *write*, respectively. The read $a[i]$ returns the value stored at position i of a, while the write $a\{i \leftarrow e\}$ is the array a modified so that position i has value e. For multidimensional arrays, we abbreviate $a[i] \cdots [j]$ with $a[i, \ldots, j]$.

The theory of equality with uninterpreted functions (EUF), T_{EUF}, is used in the decision procedure.

Definition 2 (Terms and Sorts). Index variables and terms have sort $\mathbb{Z}$ and are Presburger arithmetic terms. Element variables and terms have sort elem_k, for some element theory T^k_{elem}. Array variables and terms have functional sorts constructed from the $\mathbb{Z}$ and elem_k sorts:

– *One-dimensional sort*: $\mathbb{Z} \to \mathsf{elem}_k$, for some element theory T^k_{elem}
– *Multidimensional sort*: $\mathbb{Z} \to \cdots \to \mathsf{elem}_k$, for some element theory T^k_{elem}; *e.g.*, a two-dimensional array has sort $\mathbb{Z} \to \mathbb{Z} \to \mathsf{elem}_k$

For element term e, both a and $a\{i \leftarrow e\}$ are array terms; the latter term is a with position i modified to e. For array term a and index term i, $a[i]$ is either an element term if a has sort $\mathbb{Z} \to \mathsf{elem}_k$, or an array term if a has a multidimensional sort; *e.g.*, if a has sort $\mathbb{Z} \to \mathbb{Z} \to \mathsf{elem}_k$, then $a[i, j]$ is an element term of sort elem_k, while $a[i]$ is an array term of sort $\mathbb{Z} \to \mathsf{elem}_k$.

Definition 3 (Literal and Formula). A T_{A}-literal is either a $T_{\mathbb{Z}}$-literal or a T^k_{elem}-literal; literals can contain array subterms. A *formula* ψ in T_{A} is a quantified Boolean combination of T_{A}-literals.

Notationally, we say $\psi[t]$ is the formula that contains subterm t. $t \in \psi$ is true iff ψ contains subterm t.

We study satisfiability for a fragment of T_{A} that is a subset of the $\exists^*\forall^*_{\mathbb{Z}}$-fragment of T_{A}, where the subscript on $\forall$ indicates that the quantifier is only over index variables. We call this fragment the *array property* fragment.

Definition 4 (Array Property). An *array property* is a formula of the form

$$(\forall \bar{i})(\varphi_I(\bar{i}) \to \varphi_V(\bar{i}))$$

where $\bar{i}$ is a vector of index variables, and $\varphi_I(\bar{i})$ and $\varphi_V(\bar{i})$ are the *index guard* and the *value constraint*, respectively. The *height* of the property is the number of quantified index variables in the formula.

The form of an *index guard* $\varphi_I(\bar{i})$ is constrained according to the grammar

$$\begin{array}{rcl}
\mathsf{iguard} & \to & \mathsf{iguard} \wedge \mathsf{iguard} \mid \mathsf{iguard} \vee \mathsf{iguard} \mid \mathsf{atom} \\
\mathsf{atom} & \to & \mathsf{expr} \leq \mathsf{expr} \mid \mathsf{expr} = \mathsf{expr} \\
\mathsf{expr} & \to & uvar \mid \mathsf{pexpr} \\
\mathsf{pexpr} & \to & \mathbb{Z} \mid \mathbb{Z} \cdot evar \mid \mathsf{pexpr} + \mathsf{pexpr}
\end{array}$$

where *uvar* is any universally quantified variable, and *evar* is any existentially quantified integer variable.

The form of a *value constraint* $\varphi_V(\bar{i})$ is also constrained. Any occurrence of a quantified index variable $i \in \bar{i}$ in $\varphi_V(\bar{i})$ must be as a read into an array, $a[i]$, for array term a. Array reads may not be nested; *e.g.*, $a_1[a_2[i]]$ is not allowed.

Definition 5 (Array Property Fragment). The *array property fragment* of T_{A} consists of all existentially-closed Boolean combinations of array property formulae and quantifier-free T_{A}-formulae. The *height* of a formula in the fragment is the maximum height of an array property subformula.

Example 1 (Equality Predicates). Extensionality can be encoded in the array property fragment. We present $=$ and *bounded equality* as defined predicates. In the satisfiability decision procedure, instances of defined predicates are expanded to their definitions in the first step.

Equality. $a = b$: Arrays a and b are equal.

$$(\forall i)(a[i] = b[i])$$

Bounded Equality. $\mathsf{beq}(\ell, u, a, b)$: Arrays a and b are equal in the interval $[\ell, u]$.

$$(\forall i)(\ell \leq i \leq u \ \rightarrow\ a[i] = b[i])$$

Example 2 (Sorting Predicates). More specialized predicates can also be defined in the array property fragment. Consider the following predicates for specifying properties useful for reasoning about sortedness of integer arrays in the array property fragment of $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$.

Sorted. $\mathsf{sorted}(\ell, u, a)$: Integer array a is sorted (nondecreasing) between elements ℓ and u.

$$(\forall i, j)(\ell \leq i \leq j \leq u \ \rightarrow\ a[i] \leq a[j])$$

Partitioned. $\mathsf{partitioned}(\ell_1, u_1, \ell_2, u_2, a)$: Integer array a is partitioned such that all elements in $[\ell_1, u_1]$ are less than or equal to every element in $[\ell_2, u_2]$.

$$(\forall i, j)(\ell_1 \leq i \leq u_1 < \ell_2 \leq j \leq u_2 \ \rightarrow\ a[i] \leq a[j])$$

The literal $u_1 < \ell_2$ can be expressed as $u_1 \leq \ell_2 - 1$ so that the syntactic restrictions are met.

Example 3 (Array Property Formula). The following formula is in the array property fragment of $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$:

$$(\exists \ \mathsf{array}\ a)(\exists w, x, y, z, k, \ell, n \in \mathbb{Z}) \left[\begin{array}{c} w < x < y < z \ \wedge\ 0 < k < \ell < n \ \wedge\ \ell - k > 1 \\ \wedge\ \mathsf{sorted}(0, n-1, a\{k \leftarrow w\}\{\ell \leftarrow x\}) \ \wedge\ \mathsf{sorted}(0, n-1, a\{k \leftarrow y\}\{\ell \leftarrow z\}) \end{array}\right].$$

3 Decision Procedure $\mathsf{SAT_A}$

We now define the decision procedure $\mathsf{SAT_A}$ for satisfiability of formulae from the array property fragment. After removing array writes and skolemizing existentially quantified variables, $\mathsf{SAT_A}$ rewrites universally quantified subterms to finite conjunctions by instantiating the quantified variables over a set of index terms. The next definitions construct the set of index terms that is sufficient for making this procedure complete.

Definition 6 (Read Set). The *read set* for formula ψ is the set

$$\mathcal{R} \stackrel{\text{def}}{=} \{t \ : \ \cdot[t] \in \psi\}$$

for t representing index terms that are not universally quantified index variables.

Definition 7 (Bounds Set). The *bounds set* $\mathcal{B}$ for formula ψ is the set of Presburger arithmetic terms that arise as root pexprs (*i.e.*, pexpr terms whose parent is an expr) during the parsing of all index guards in ψ, according to the grammar of Def. 4.

The read set $\mathcal{R}$ is the set of index terms at which some array is read, while the bounds set $\mathcal{B}$ is the set of index terms that define boundaries on some array for an array property (*e.g.*, the boundaries of an interval in which array elements are sorted).

Definition 8 (Index Set). For a formula ψ, define

$$\mathcal{I}_\psi \stackrel{\text{def}}{=} \begin{cases} \{0\} & \text{if } \mathcal{R} = \mathcal{B} = \emptyset \\ \mathcal{R} \cup \mathcal{B} & \text{otherwise} \end{cases}$$

The procedure reduces the satisfiability of array property formula ψ to the satisfiability of a quantifier-free $(T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T^k_{\mathsf{elem}})$-formula.

Definition 9 ($\mathsf{SAT_A}$).

1. Replace instances of defined predicates with their definitions, and convert to negation normal form.
2. Apply the following rule exhaustively to remove writes:

$$\frac{\psi[a\{i \leftarrow e\}]}{\psi[b] \ \wedge \ b[i] = e \ \wedge \ (\forall j)(j \neq i \ \rightarrow \ a[j] = b[j])} \text{ for fresh } b \qquad \text{(write)}$$

   To meet the syntactic requirements on an index guard, we rewrite the third conjunct as

$$(\forall j)(j \leq i - 1 \ \vee \ i + 1 \leq j \ \rightarrow \ a[j] = b[j]) \ .$$

3. Apply the following rule exhaustively:

$$\frac{\psi[(\exists \bar{i})(\varphi_I(\bar{i}) \wedge \neg\varphi_V(\bar{i}))]}{\psi[\varphi_I(\bar{j}) \wedge \neg\varphi_V(\bar{j})]} \text{ for fresh } \bar{j} \qquad \text{(exists)}$$

4. Apply the following rule exhaustively, where $\mathcal{I}_{\psi_3}$ is determined by the formula constructed in Step 3.

$$\frac{\psi[(\forall \bar{i})(\varphi_I(\bar{i}) \rightarrow \varphi_V(\bar{i}))]}{\psi\left[\bigwedge_{\bar{i} \in \mathcal{I}^n_{\psi_3}} (\varphi_I(\bar{i}) \rightarrow \varphi_V(\bar{i}))\right]} \qquad \text{(forall)}$$

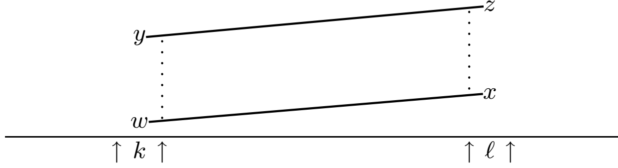


Fig. 1. Unsorted arrays

5. Associate with each n-dimensional array variable a a fresh n-ary uninterpreted function f_a, and replace each array read $a[i, \ldots, j]$ by $f_a(i, \ldots, j)$. Decide this formula's satisfiability using a procedure for quantifier-free formulae of $T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T^k_{\mathsf{elem}}$.

Step 2 introduces new index terms ($i - 1$ and $i + 1$, above).

Example 4 (New Indices). Consider again the array property formula

$$\begin{array}{c} w < x < y < z \ \wedge \ 0 < k < \ell < n \ \wedge \ \ell - k > 1 \\ \wedge \ \mathsf{sorted}(0, n-1, a\{k \leftarrow w\}\{\ell \leftarrow x\}) \ \wedge \ \mathsf{sorted}(0, n-1, a\{k \leftarrow y\}\{\ell \leftarrow z\}) \end{array}$$

(which is existentially closed). The first step of $\mathsf{SAT_A}$ replaces the sorted literals with definitions; the second applies write to remove array writes. For readability, we write the index guards resulting from write using disequalities:

$$\begin{array}{l} \ \ w < x < y < z \ \wedge \ 0 < k < \ell < n \ \wedge \ \ell - k > 1 \\ \wedge \ (\forall i, j)(0 \leq i \leq j \leq n-1 \ \rightarrow \ c[i] \leq c[j]) \\ \wedge \ (\forall i, j)(0 \leq i \leq j \leq n-1 \ \rightarrow \ e[i] \leq e[j]) \\ \wedge \ (\forall i)(i \neq \ell \ \rightarrow \ b[i] = c[i]) \ \wedge \ c[\ell] = x \\ \wedge \ (\forall i)(i \neq k \ \rightarrow \ a[i] = b[i]) \ \wedge \ b[k] = w \\ \wedge \ (\forall i)(i \neq \ell \ \rightarrow \ d[i] = e[i]) \ \wedge \ e[\ell] = z \\ \wedge \ (\forall i)(i \neq k \ \rightarrow \ a[i] = d[i]) \ \wedge \ d[k] = y \end{array}$$

Then $\mathcal{R} = \{k, \ell\}$, $\mathcal{B} = \{0, n-1, \ell-1, \ell+1, k-1, k+1\}$, and $\mathcal{I}_\psi = \{0, n-1, k-1, k, k+1, \ell-1, \ell, \ell+1\}$. Note that $\mathcal{R}$ and $\mathcal{B}$ do not include i or j, which are universally quantified, while $\mathcal{B}$ contains the terms produced by converting disequalities to disjunctions of inequalities. Applying forall to each array property subformula converts universal quantification to finite conjunction over $\mathcal{I}_\psi$. We have in particular that

$$c[k+1] \leq c[\ell] = x < y = d[k] \leq d[k+1] \ ,$$

yet $c[k+1] = b[k+1] = a[k+1] = d[k+1]$, a contradiction. Thus, the original formula is $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$-unsatisfiable. The index term $k+1$ is essential for this proof.

We visualize this situation in Figure 1. Arrows indicate positions represented by the new indices introduced in Step 2. Pictorially, for both modified versions of a to be sorted requires that the two parallel lines in Figure 1 be one line. To prove that sortedness is impossible requires considering elements in the interval between k and ℓ, not just elements at positions k and ℓ.

4 Correctness

We prove the soundness and completeness of $\mathsf{SAT_A}$. Additionally, we show that if satisfiability of quantifier-free $(T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T^k_{\mathsf{elem}})$-formulae is in NP, then satisfiability for each *bounded* fragment, in which all array properties have maximum height N, is NP-complete.

We refer to the formula constructed in Step n of $\mathsf{SAT_A}$ by ψ_n; *e.g.*, ψ_5 is the final quantifier-free formula constructed in Step 5.

Lemma 1 (Complete). *If ψ_5 is satisfiable, then ψ is satisfiable.*

Proof. Suppose that I is an interpretation such that $I \models \psi_5$; we construct an interpretation J such that $J \models \psi$. To this end, we define under I a *projection* operation, $\mathsf{proj} : \mathbb{Z} \to \mathcal{I}^I_{\psi_3}$: $\mathsf{proj}(z) = t^I$ such that $t \in \mathcal{I}_{\psi_3}$; and either $t^I \leq z$ and $(\forall s \in \mathcal{I}_{\psi_3})(s^I \leq t^I \ \vee\ s^I > z)$, or $t^I > z$ and $(\forall s \in \mathcal{I}_{\psi_3})(s^I \geq t^I)$. That is, $\mathsf{proj}(z)$ is the nearest neighbor to z in t^I, with preference for left neighbors. Extend proj to tuples of integers in the natural way: $\mathsf{proj}(z_1, \ldots, z_k) = (\mathsf{proj}(z_1), \ldots, \mathsf{proj}(z_k))$.

Equate all non-array variables in J and I; note that proj is now defined the same under I and J. For each k-dimensional array a of ψ, set $a^J[\overline{z}] = f^J_a(\mathsf{proj}(\overline{z}))$. We now prove that $J \models \psi$.

The manipulations in Steps 1, 3, and 5 are trivial. Step 2 implements the definition of array write, so that the resulting formula is equivalent to the original formula. Thus, we focus on Step 4. We prove that if $J \models \psi_4$, then $J \models \psi_3$.

Suppose that rule forall is applied to convert ψ_b to ψ_a and that $J \models \psi_a$. Application of this rule is the main focus of the proof: we prove that $J \models \psi_b$. That is, we assume that

$$J \models \underbrace{\psi' \left[\bigwedge_{\overline{i} \in \mathcal{I}^n_\psi} (\varphi_I(\overline{i}) \to \varphi_V(\overline{i})) \right]}_{\psi_a} \tag{1}$$

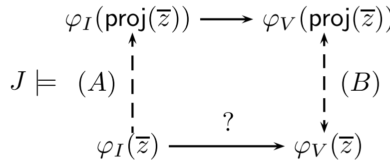
and prove that

$$J \models \underbrace{\psi' \left[(\forall \overline{i})(\varphi_I(\overline{i}) \to \varphi_V(\overline{i})) \right]}_{\psi_b} . \tag{2}$$

Below, we prove that

$$J \models \left[\bigwedge_{\overline{i} \in \mathcal{I}^n_\psi} (\varphi_I(\overline{i}) \to \varphi_V(\overline{i})) \ \to \ (\forall \overline{i})(\varphi_I(\overline{i}) \to \varphi_V(\overline{i})) \right] ,$$

which implies (2) since ψ' is in negation normal form. Our proof takes the form



where, for arbitrary $\overline{z} \in \mathbb{Z}^n$, we prove the implication labeled "?" by proving (A) and (B). The top implication follows from (1) and the definition of proj.

For (A), consider the atoms of the index guard under J. If $\ell^J \leq m^J$, then the definition of proj implies that $\mathsf{proj}(\ell^J) \leq \mathsf{proj}(m^J)$. At worst, it may be that $\ell^J < m^J$, while $\mathsf{proj}(\ell^J) = \mathsf{proj}(m^J)$. For an equation, $\ell^J = m^J$ iff $\mathsf{proj}(\ell^J) = \mathsf{proj}(m^J)$. Then (A) follows by structural induction over the index guard, noting that the index guard is a positive Boolean combination of atoms.

For (B), recall that arrays in J are constructed using proj. In particular, for any $\overline{z}$, $a^J[\overline{z}] = a^J[\mathsf{proj}(\overline{z})]$, so that (B) follows.

Therefore, $J \models \psi_b$, and $\mathsf{SAT_A}$ is complete.

Lemma 2 (Sound). *If ψ is satisfiable, then ψ_5 is satisfiable.*

Proof. An interpretation I satisfying ψ can be altered to J satisfying ψ_5 by assigning $f_a^J(\overline{i}^I) = a^I[\overline{i}^I]$ for each array variable a and equating all else. Universal quantification is replaced by conjunction over a finite subset of all indices, thus weakening each (positive) literal.

Theorem 1. *If satisfiability of quantifier-free $(T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T_{\mathsf{elem}}^k)$-formulae is decidable, then $\mathsf{SAT_A}$ is a decision procedure for satisfiability in the array property fragment of $T_{\mathsf{A}}^{\{\mathsf{elem}_k\}_k}$.*

Theorem 2 (NP-Complete). *If satisfiability of quantifier-free $(T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T_{\mathsf{elem}}^k)$-formulae is in NP, then for the subfragment of the array property fragment of $T_{\mathsf{A}}^{\{\mathsf{elem}_k\}_k}$ in which all array property formulae have height at most N, satisfiability is NP-complete.*

Proof. NP-hardness, even when ψ is a conjunction of literals, follows by NP-hardness of satisfiability of $T_{\mathbb{Z}}$ [6]. Steps 1-3 increase the size of the formula by an amount linear in the size of ψ. The rule forall increases the size of formulae by an amount polynomial in the size of ψ and exponential in the maximum height N. For fixed N, the increase is thus polynomial in ψ. The proof requires only a polynomial number (in the size of ψ) of applications of rules, so that the size of the quantifier-free $(T_{\mathsf{EUF}} \cup T_{\mathbb{Z}} \cup \bigcup_k T_{\mathsf{elem}}^k)$-formula is at most polynomially larger than ψ. Inclusion in NP follows from the assumption of the theorem.

5 Undecidable Problems

Theorem 1 states that for certain sets of element theories $\{\mathsf{elem}_k\}_k$, $\mathsf{SAT_A}$ is a satisfiability decision procedure for the array property fragment of $T_{\mathsf{A}}^{\{\mathsf{elem}_k\}_k}$. The theory of reals, $T_{\mathbb{R}}$, in which variables range over $\mathbb{R}$ and with signature $\Sigma_{\mathbb{R}} = \{0, 1, +, -, =, <\}$, and the theory of integers, $T_{\mathbb{Z}}$, are such element theories. We now show that several natural extensions of the array property fragment result in a fragment of $T_{\mathsf{A}}^{\{\mathbb{R}\}}$ or $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$ for which satisfiability is undecidable. We identify one extension for which decidability remains open.

Theorem 3. *Satisfiability of the $\exists^*\forall_{\mathbb{Z}}\exists_{\mathbb{Z}}$-fragment of both $T_{\mathsf{A}}^{\{\mathbb{R}\}}$ and $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$ is undecidable, even with syntactic restrictions like in the array property fragment.*

Proof. In [3], we prove that termination of loops of this form is undecidable:

$$\begin{array}{l}
\textbf{real } x_1, \ldots, x_n \\
\theta : \displaystyle\bigwedge_{i \in I \subseteq \{1,\ldots,n\}} x_i = c_i \\
\textbf{while } x_1 \geq 0 \textbf{ do} \\
\quad \textbf{choose } \tau_i : \ \mathbf{x} := A_i \mathbf{x} \\
\textbf{done}
\end{array}$$

c_i are constant integers, $c_i \in \mathbb{Z}$, while each A_i is an $n \times n$ constant array of integers, $A_i \in \mathbb{Z}^{n \times n}$. θ is the initial condition of the loop. Variables $x_1, \ldots, x_n$ range over the reals, $\mathbb{R}$. There are $m > 0$ transitions, $\{\tau_1, \ldots, \tau_m\}$; on each iteration, one is selected nondeterministically to be taken. $\mathbf{x}$ is an $\mathbb{R}^n$-vector representing the n variables $\{x_1, \ldots, x_n\}$; each transition thus updates all variables simultaneously by a linear transformation. We call loops of this form linear loops. Termination for similar loops in which all variables are declared as integers is also undecidable.

We now prove by reduction from termination of linear loops that satisfiability of the $\exists^*\forall_{\mathbb{Z}}\exists_{\mathbb{Z}}$-fragment is undecidable. That is, given linear loop L, we construct formula φ such that φ is unsatisfiable iff L always terminates. In other words, a model of φ encodes a nonterminating computation of L.

For each loop variable x_i, we introduce array variable x_i. Let $\rho_\tau(s,t)$, for index terms s and t, encode transition $\tau : \ \mathbf{x} := A\mathbf{x}$ as follows:

$$\rho_\tau(s,t) \ \stackrel{\text{def}}{=} \ \bigwedge_{i=1}^{n} x_i[t] = A_{i,1} \cdot x_1[s] + \cdots + A_{i,n} \cdot x_n[s] \ .$$

Let $g(s)$, for index term s, encode the guard $x_1 \geq 0$:

$$g(s) \ \stackrel{\text{def}}{=} \ x_1[s] \geq 0 \ .$$

Let $\theta(s)$, for index term s, encode the initial condition:

$$\theta(s) \ \stackrel{\text{def}}{=} \bigwedge_{i \in I \subseteq \{1,\ldots,n\}} x_i[s] = c_i \ .$$

Then form φ:

$$\varphi : \ (\exists x_1, \ldots, x_n, z)(\forall i)(\exists j) \left[\theta(z) \ \wedge \ g(z) \ \wedge \ \bigvee_k \rho_{\tau_k}(i,j) \ \wedge \ g(j) \right] \ .$$

Suppose φ is satisfiable. Then construct a nonterminating computation $s_0 s_1 s_2 \ldots$ as follows. Let each variable x_k of state s_0 take on the value $x_k[z]$

of the satisfying model. For s_1, choose the j that corresponds to $i = z$ and assign x_k according to $x_k[j]$. Continue forming the computation sequentially. Each state is guaranteed to satisfy the guard, so the computation is nonterminating.

Suppose $s_0 s_1 s_2 \ldots$ is a nonterminating computation of L. Then construct the following model for φ. Let $z = 0$; for each index $i \geq 0$, set $x_k[-i] = x_k[i] = x_k$ of state s_i.

Therefore, φ is unsatisfiable iff L always terminates, and thus satisfiability of the $\exists^* \forall_{\mathbb{Z}} \exists_{\mathbb{Z}}$-fragment of T_{A} is undecidable. Note that φ meets the syntactic restrictions of the array property fragment, except for the extra quantifier alternation.

Theorem 4. *Extending the array property fragment with any of*

- *nested reads (e.g., $a_1[a_2[i]]$, where i is universally quantified);*
- *array reads by a universally quantified variable in the index guard;*
- *general Presburger arithmetic expressions over universally quantified index variables (even just addition of 1, e.g., $i + 1$) in the index guard or in the value constraint*

results in a fragment of $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$ for which satisfiability is undecidable.

Proof. In $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$, the presence of nested reads allows skolemizing j in φ of the proof of Theorem 3:

$$(\exists x_1, \ldots, x_n, z, a_j)(\forall i) \left[\theta(z) \ \wedge \ g(z) \ \wedge \ \bigvee_k \rho_{\tau_k}(i, a_j[i]) \ \wedge \ g(a_j[i]) \right] .$$

Allowing array reads in the index guard enables flattening of nested reads through introduction of another universally quantified variable:

$$\psi[\varphi_I \ \rightarrow \ \varphi_V[a[a[i]]]] \quad \Rightarrow \quad \psi[\varphi_I \wedge j = a[i] \ \rightarrow \ \varphi_V[a[j]]] .$$

Allowing addition of 1 in the value constraint allows an encoding of termination similar to that in the proof of Theorem 3:

$$(\exists x_1, \ldots, x_n, z)(\forall i \geq z) \left[\theta(z) \ \wedge \ g(z) \ \wedge \ \bigvee_k \rho_{\tau_k}(i, i+1) \ \wedge \ g(i+1) \right] .$$

Finally, addition of 1 in the index guard can encode addition of 1 in the value constraint through introduction of another universally quantified variable:

$$\psi[\varphi_I \ \rightarrow \ \varphi_V[a[i+1]]] \quad \Rightarrow \quad \psi[\varphi_I \wedge j = i + 1 \ \rightarrow \ \varphi_V[a[j]]] .$$

Theorem 3 implies that a negated array property cannot be embedded in the consequent of another array property. Theorem 4 states that loosening most syntactic restrictions results in a fragment for which satisfiability is undecidable. One extension remains for which decidability of satisfiability is an open problem: the fragment in which index guards can contain strict inequalities, $<$ (equivalently, in which index guards can contain negations). In this fragment, one could express that an array has unique elements:

$$(\forall i, j)(i < j \ \rightarrow \ a[i] \neq a[j]) .$$

6 Maps

We consider an array theory in which indices are uninterpreted. For clarity, we call indices *keys* in this theory, and call the arrays *maps*.

Definition 10 (Map Theory). The parameterized map theory $T_{\mathsf{M}}^{\{\mathsf{elem}_\ell\}_\ell}$ has signature

$$\Sigma_{\mathsf{M}} = \Sigma_{\mathsf{EUF}} \cup \bigcup_{\ell} \Sigma^{\ell}_{\mathsf{elem}} \cup \{\cdot[\cdot], \cdot\{\cdot \leftarrow \cdot\}\} .$$

Key variables and terms are uninterpreted, with sort EUF. Element variables and terms have some sort elem_ℓ. Map variables and terms have functional sorts constructed from the EUF and elem_ℓ sorts; *e.g.*, $\mathsf{EUF} \rightarrow \mathsf{elem}_\ell$.

Definition 11 (Map Property Fragment). A *map property* is a formula of the form $(\forall \overline{k})(\varphi_K(\overline{k}) \rightarrow \varphi_V(\overline{k}))$, where $\overline{k}$ is a vector of key variables, and $\varphi_K(\overline{k})$ and $\varphi_V(\overline{k})$ are the *key guard* and the *value constraint*, respectively. The *height* of the property is the number of quantified variables in the formula.

The form of a *key guard* $\varphi_K(\overline{k})$ is constrained according to the grammar

$$\begin{array}{rcl} \mathsf{kguard} & \rightarrow & \mathsf{kguard} \wedge \mathsf{kguard} \mid \mathsf{kguard} \vee \mathsf{kguard} \mid \mathsf{atom} \\ \mathsf{atom} & \rightarrow & \mathsf{var} = \mathsf{var} \mid \mathit{evar} \neq \mathsf{var} \mid \mathsf{var} \neq \mathit{evar} \\ \mathsf{var} & \rightarrow & \mathit{evar} \mid \mathit{uvar} \end{array}$$

where *uvar* is any universally quantified key variable, and *evar* is any existentially quantified variable.

The form of a *value constraint* $\varphi_V(\overline{k})$ is also constrained. Any occurrence of a quantified key variable $k \in \overline{k}$ in $\varphi_V(\overline{k})$ must be as a read into a map, $h[k]$, for map term h. Map reads may not be nested; *e.g.*, $h_1[h_2[k]]$ is not allowed.

The *map property fragment* of T_{M} consists of all existentially-closed Boolean combinations of map property formulae and quantifier-free T_{M}-formulae.

Definition 12 (Key Set). 2 For a formula ψ, define $\mathcal{R} = \{t \; : \; \cdot[t] \in \psi\}$; $\mathcal{B}$ as the set of variables that arise as *evar*s in the parsing of all key guards according to the grammar of Def. 11; and $\mathcal{K} = \mathcal{R} \cup \mathcal{B} \cup \{\kappa\}$, where κ is a fresh variable.

Definition 13 ($\mathsf{SAT_M}$).

1. Step 1 of $\mathsf{SAT_A}$.
2. Apply the following rule exhaustively to remove writes:

$$\frac{\psi[h\{k \leftarrow e\}]}{\psi[h'] \;\wedge\; h'[k] = e \;\wedge\; (\forall j)(j \neq k \;\rightarrow\; h[j] = h'[j])} \text{ for fresh } h' \qquad (\mathsf{write})$$

3. Step 3 of $\mathsf{SAT_A}$.

4. Apply the following rule exhaustively, where $\mathcal{K}_{\psi_3}$ is determined by the formula constructed in Step 3.

$$\frac{\psi[(\forall \overline{k})(\varphi_K(\overline{k}) \rightarrow \varphi_V(\overline{k}))]}{\psi\left[\bigwedge_{\overline{k} \in \mathcal{K}^n_{\psi_3}} (\varphi_K(\overline{k}) \rightarrow \varphi_V(\overline{k}))\right]} \quad \text{(forall)}$$

5. Construct

$$\psi_4 \;\wedge \bigwedge_{k \in \mathcal{K} \setminus \{\kappa\}} k \neq \kappa$$

6. Step 5 of $\mathsf{SAT_A}$, except that the resulting formula is decided using a procedure for $T_{\mathsf{EUF}} \cup \bigcup_\ell T^\ell_{\mathsf{elem}}$.

Theorem 5. *If satisfiability of quantifier-free* $(T_{\mathsf{EUF}} \cup \bigcup_\ell T^\ell_{\mathsf{elem}})$*-formulae is decidable, then* $\mathsf{SAT_M}$ *is a decision procedure for satisfiability in the map property fragment of* $T_{\mathsf{M}}^{\{\mathsf{elem}_\ell\}_\ell}$.

The main idea of the proof, as in the proof of Theorem 1, is to define a projection operation, $\mathsf{proj} : \mathsf{EUF} \rightarrow \mathcal{K}^I_{\psi_3}$, for interpretation I. For object o of I, if $o = t^I$ for some $t \in \mathcal{K}_{\psi_3}$, then $\mathsf{proj}(o) = o \; (= t^I)$; otherwise, $\mathsf{proj}(o) = \kappa^I$. If proj is used to define J, as in the proof of Theorem 1, then proj preserves equations and disequalities in key guards and values of map reads in value constraints.

The relevant undecidability results from Section 5 carry over to maps, with the appropriate modifications.

7 Applications, Implementation, and Results

7.1 Verification of Sorting Algorithms

Figure 2 presents an annotated version of InsertionSort in an imperative language, where the annotations specify that InsertionSort returns a sorted array. @pre, @post, and @ label preconditions, postconditions, and (loop) assertions, respectively. For variable x, x_0 refers to its value upon entering a function; $|a|$ maps array a to its length; rv is the value returned by a function. Each verification condition is expressible in the array property fragment of $T_{\mathsf{A}}^{\{\mathbb{Z}\}}$ and is unsatisfiable, proving that InsertionSort returns a sorted array.

7.2 Verification of Parameterized Programs

The parallel composition of an arbitrary number of copies of a process is often represented as a *parameterized program*. Variables for which one copy appears in each process are modeled as arrays. Thus, it is natural to specify and prove properties of parameterized programs with a language of arrays.

```
@pre ⊤
@post sorted(0, |rv| − 1, rv)
int[] InsertionSort(int[] a) {
  int i, j, t;
  for (i := 1; i < |a|; i := i + 1)
     @(1 ≤ i ∧ sorted(0, i − 1, a) ∧ |a| = |a_0|)
  {
    t := a[i];
    for (j := i − 1; j ≥ 0 ∧ a[j] > t; j := j − 1)
       @[   1 ≤ i < |a| ∧ −1 ≤ j ≤ i − 1
         ∧ sorted(0, i − 1, a) ∧ |a| = |a_0|
         ∧ (j < i − 1 → (a[i − 1] ≤ a[i] ∧ (∀k ∈ [j + 1, i]) a[k] > t)) ]
      a[j + 1] := a[j];
    a[j + 1] := t;
  }
  return a;
}
```

Fig. 2. InsertionSort

```
int[] y := int[0..M − 1];
θ: y[0] = 1 ∧ (∀j ∈ [1, M − 1]) y[j] = 0

    ||       [ request(y, i);
i∈[0,M−1]      while (true) @((∀j ∈ [0, M − 1]) y[j] = 0 ∧ i = i_0 ∧ |y| = |y_0|) {
                 critical;
                 release(y, i ⊕_M 1);
                 noncritical;
                 request(y, i);
               }                                                              ]
```

Fig. 3. Sem-N

Figure 3 presents a simple semaphore-based algorithm for mutual exclusion among M processes [4]. The semantics of **request** and **release** are the usual ones:

$$\begin{aligned} \textbf{request}(y,i): &\quad y[i] > 0 \ \wedge \ y' = y\{i \leftarrow y[i]-1\} \\ \textbf{release}(y,i): &\quad y' = y\{i \leftarrow y[i]+1\} \end{aligned}$$

Mutual exclusion at the **critical** section is implied by the invariant $(\forall j \in [0, M-1])\ y[j] = 0$, which appears as part of the loop invariant. The mutual exclusion property is verified using the array decision procedure.

7.3 A Decision Procedure for Hashtables

We show how to encode an assertion language for hashtables, with parameter theories $T^1_{\mathsf{elem}}, \ldots, T^m_{\mathsf{elem}}$ for values, into $T_{\mathsf{M}}^{\{\mathsf{elem}_\ell\}_\ell}$. Hashtables have the following operations: $\mathsf{put}(h, k, v)$ returns the hashtable that is equal to h except that key k maps to value v; $\mathsf{remove}(h, k)$ returns the hashtable that is equal to h except that key k does not map to a value; and $\mathsf{get}(h, k)$ returns the value mapped by k,

which is undetermined if h does not map k to any value. $\mathsf{init}(h)$ is true iff h does not map any key. For reasoning about keys, $k \in \mathsf{keys}(h)$ is true iff h maps k; key sets $\mathsf{keys}(h)$ can be unioned, intersected, and complemented. For the encoding onto the map property fragment of T_{M}, universal quantification is restricted to quantification over key variables; such variables may only be used in membership checking, $k \in K$, and gets, $\mathsf{get}(h, k)$. Finally, an init in the scope of a universal quantifier must appear positively. The encoding then works as follows:

1. Construct $\psi \;\wedge\; \top \neq \bot$, for fresh constants $\top$ and $\bot$.
2. Rewrite

$$\begin{array}{rcl} \psi[\mathsf{put}(h,k,v)] & \Rightarrow & \psi[h'] \;\wedge\; h' = h\{k \leftarrow v\} \;\wedge\; \mathsf{keys}_{h'} = \mathsf{keys}_h\{k \leftarrow \top\} \\ \psi[\mathsf{remove}(h,k)] & \Rightarrow & \psi[h'] \;\wedge\; \mathsf{keys}_{h'} = \mathsf{keys}_h\{k \leftarrow \bot\} \end{array}$$

   for fresh variable h'.
3. Rewrite

$$\begin{array}{rcl} \psi[\mathsf{get}(h,k)] & \Rightarrow & \psi[h[k]] \\ \psi[\mathsf{init}(h)] & \Rightarrow & \psi[(\forall k)(h[k] = \bot)] \\ \psi[k \in \mathsf{keys}(h)] & \Rightarrow & \psi[\mathsf{keys}_h[k] \neq \bot] \\ \psi[k \in K_1 \cup K_2] & \Rightarrow & \psi[k \in K_1 \;\vee\; k \in K_2] \\ \psi[k \in K_1 \cap K_2] & \Rightarrow & \psi[k \in K_1 \;\wedge\; k \in K_2] \\ \psi[k \in \overline{K}] & \Rightarrow & \psi[\neg(k \in K)] \end{array}$$

   where K, K_1, and K_2 are constructed from union, disjunction, and complementation of membership atoms.

Note that we rely on the defined predicate of equality between maps, $h_1 = h_2$, which is defined by $(\forall k)(h_1[k] = h_2[k])$. Subset checking between key sets, $K_1 \subset K_2$, and other useful operations can also be defined in this language.

An example specification might assert that $(\forall k \in \mathsf{keys}(h))(\mathsf{get}(h,k) \geq 0)$. Suppose that a function modifies h; then a verification condition could be

$$(\forall h, s, v, h') \left[\begin{array}{l} (\forall k \in \mathsf{keys}(h))\; \mathsf{get}(h,k) \geq 0 \;\wedge\; v \geq 0 \;\wedge\; h' = \mathsf{put}(h,s,v) \\ \quad \rightarrow\; (\forall k \in \mathsf{keys}(h'))\; \mathsf{get}(h',k) \geq 0 \end{array} \right].$$

The key sets provide a mechanism for reasoning about modifying hashtables.

7.4 Implementation and Results

We implemented $\mathsf{SAT_A}$ in our verifying compiler, $\pi\mathsf{VC}$, which verifies programs written in the pi (for *Prove It*) programming language. The syntax of the language is similar to that of Figure 2. We used CVC Lite [2] as the underlying decision procedure. We found that there is usually no need to instantiate quantifiers with all terms in $\mathcal{I}$; instead, the implementation makes several attempts to prove a formula unsatisfiable. It first tries using the set $\mathcal{R}$, then $\mathcal{R} \cup \mathcal{B}$, and finally $\mathcal{I}$. Moreover, common sense rules restrict the instantiation in the early attempts. If any attempt results in an unsatisfiable formula, then the original formula is unsatisfiable; and if the formula of the final attempt is satisfiable, then the original formula is satisfiable.

Frame conditions are ubiquitous in verification conditions. Thus, the implementation performs a simple form of resolution to simplify the original formula. After rewriting based on equations in the antecedent, conjuncts in the consequent that are syntactically equal to conjuncts in the antecedent are replaced with true. In practice, the resulting index sets are smaller. The combination of the phased instantiation and simplification makes the decision procedure quite responsive in practice.

We implemented annotated versions of MergeSort, BubbleSort, InsertionSort, QuickSort, Sem-N, and BinarySearch for integer arrays in our programming language pi. Verifying that the sorting algorithms return sorted arrays required less than 20 seconds each (1 second for each of BubbleSort and InsertionSort). Verifying mutual exclusion in Sem-N required a second. Verifying the membership property of BinarySearch required a second. All tests were performed on a 3 GHz x86; memory was not an issue.

8 Future Work

Future work will focus on the decidability of the extension identified in Section 5; on the complexity of deciding satisfiability for the full array property fragment for particular element theories; and, most importantly, on generating inductive invariants in the array property fragment automatically.

Acknowledgments. We thank the reviewers, Tom Henzinger, and members of the STeP group for their insightful comments and suggestions on this work.

References

1. Armando, A., Ranise, S., and Rusinowitch, M. Uniform derivation of decision procedures by superposition. In *International Workshop on Computer Science Logic (CSL)* (2001), Springer-Verlag.
2. Barrett, C., and Berezin, S. CVC Lite: A new implementation of the cooperating validity checker. In *Computer Aided Verification (CAV)* (2004), Springer-Verlag.
3. Bradley, A. R., Manna, Z., and Sipma, H. B. Polyranking for polynomial loops. In submission; available at `http://theory.stanford.edu/~arbrad`.
4. Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer, 1995.
5. McCarthy, J. Towards a mathematical science of computation. In *IFIP Congress 62* (1962).
6. Schrijver, A. *Theory of Linear and Integer Programming.* Wiley, 1986.
7. Stump, A., Barrett, C. W., Dill, D. L., and Levitt, J. R. A decision procedure for an extensional theory of arrays. In *Logic in Computer Science (LICS)* (2001).
8. Suzuki, N., and Jefferson, D. Verification decidability of Presburger array programs. *J. ACM 27*, 1 (1980).

Author Index

Appel, Andrew W. 80
Arnold, Gilad 33

Balaban, Ittai 267
Bingham, Jesse 207
Bozzelli, Laura 65, 412
Bradley, Aaron R. 427

Chang, Bor-Yuh Evan 174
Chechik, Marsha 381
Chlipala, Adam 174
Clarke, Edmund 126
Cohen, Ariel 267
Colón, Michael A. 111
Cortesi, Agostino 313

Edelkamp, Stefan 237

Ganty, Pierre 49
Ghafari, Naghmeh 252
Gurfinkel, Arie 381

Hristova, Katia 190

Jabbar, Shahid 237
Jaffar, Joxan 17, 282

Kobayashi, Naoki 298
Kuncak, Viktor 157

La Torre, Salvatore 412
Lam, Patrick 157
Liu, Yanhong A. 190
Logozzo, Francesco 313

Manevich, Roman 33
Manna, Zohar 111, 427
Miné, Antoine 348

Necula, George C. 174

Peron, Adriano 412
Piterman, Nir 364
Pnueli, Amir 267, 364
Podelski, Andreas 157

Rakamarić, Zvonimir 207
Ranzato, Francesco 332
Raskin, Jean-François 49
Rinard, Martin 157
Rossignoli, Stefano 95

Sa'ar, Yaniv 364
Sagiv, Mooly 33
Sankaranarayanan, Sriram 111
Santosa, Andrew E. 17, 282
Schachte, Peter 1
Shaham, Ran 33
Sipma, Henny B. 111, 427
Sistla, A. Prasad 222
Søndergaard, Harald 1
Spoto, Fausto 95
Subramani, K. 398
Suenaga, Kohei 298

Talupur, Muralidhar 126
Tan, Gang 80
Tapparo, Francesco 332
Trefler, Richard 252

Van Begin, Laurent 49
Veith, Helmut 126
Voicu, Răzvan 17, 282

Wei, Ou 381
Wies, Thomas 157
Wischik, Lucian 298

Younes, Håkan L.S. 142

Zhou, Min 222
Zuck, Lenore D. 222