

# Lecture Notes in Computer Science

2725

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

**Springer**

*Berlin*

*Heidelberg*

*New York*

*Hong Kong*

*London*

*Milan*

*Paris*

*Tokyo*

Warren A. Hunt, Jr. Fabio Somenzi (Eds.)

# Computer Aided Verification

15th International Conference, CAV 2003  
Boulder, CO, USA, July 8-12, 2003  
Proceedings



Springer

## Series Editors

Gerhard Goos, Karlsruhe University, Germany  
Juris Hartmanis, Cornell University, NY, USA  
Jan van Leeuwen, Utrecht University, The Netherlands

## Volume Editors

Warren A. Hunt, Jr.  
University of Texas at Austin, Department of Computer Sciences  
Taylor Hall, M/S C0500, Austin, TX 78712-1188, USA  
E-mail: [hunt@cs.utexas.edu](mailto:hunt@cs.utexas.edu)  
Fabio Somenzi  
University of Colorado, Department of Electrical and Computer Engineering  
Campus Box 425, Boulder, CO 80309-0425, USA  
E-mail: [Fabio@Colorado.edu](mailto:Fabio@Colorado.edu)

## Cataloging-in-Publication Data applied for

Bibliographic information published by Die Deutsche Bibliothek  
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;  
detailed bibliographic data is available in the Internet at [<http://dnb.ddb.de>](http://dnb.ddb.de).

CR Subject Classification (1998): F.3, D.2.4, D.2.2, F.4.1, I.2.3, B.7.2, C.3

ISSN 0302-9743

ISBN 3-540-40524-0 Springer-Verlag 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-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York  
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2003  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Markus Richter, Heidelberg  
Printed on acid-free paper SPIN: 10929025 06/3142 5 4 3 2 1 0

# Preface

This volume contains the proceedings of the conference on *Computer Aided Verification* (CAV 2003) held in Boulder, Colorado, on July 8–12, 2003. CAV 2003 was the 15th in a series of conferences dedicated to the advancement of the theory and practice of computer-assisted formal analysis methods for hardware and software systems. The conference covers the spectrum from theoretical results to applications, with emphasis on practical verification tools, including algorithms and techniques needed for their implementation. The conference has traditionally drawn contributions from researchers as well as practitioners in both academia and industry.

The program of the conference consisted of 32 regular papers, selected from 87 submissions. In addition, the CAV program featured 9 tool presentations and demonstrations selected from 15 submissions. Each submission received an average of 5 referee reviews. The large number of tool submissions and presentations testifies to the liveliness of the field and to its applied flavor.

The CAV 2003 program included a tutorial day with three invited tutorials by Ken McMillan (Cadence) on *SAT-Based Methods for Unbounded Model Checking*, Doron Peled (Warwick) on *Algorithmic Testing Methods*, and Willem Visser (NASA) on *Model Checking Programs with Java PathFinder*. The conference also included two invited talks by Amitabh Srivastava (Microsoft) and Michael Gordon (Cambridge). Five workshops were associated with CAV 2003:

- ACL2 2003: 4th International Workshop on the ACL2 Theorem Prover and Its Applications.
- BMC 2003: 1st International Workshop on Bounded Model Checking.
- PDMC 2003: 2nd International Workshop on Parallel and Distributed Model Checking.
- RV 2003: 3rd Workshop on Runtime Verification.
- SoftMC 2003: 2nd Workshop on Software Model Checking.

The publication of these workshop proceedings was managed by the respective chairs, independently of the present proceedings.

We would like to thank all the Program Committee members and the sub-referees who assisted in the selection of the papers. Our thanks also go to the Steering Committee members and to last year's organizers for their helpful advice. Special thanks go to Virginia Schultz of the Office of Conference Services of the University of Colorado for assisting with the local arrangements; to Robert Krug for installing and managing the START Conference System; and to Erik Reeber for the production of the final proceedings. Finally, we gratefully acknowledge support from IBM, Intel, and Esterel Technologies.

## Program Committee

Rajeev Alur (Pennsylvania)	Orna Kupferman (Hebrew University)
George Avrunin (Massachusetts)	Bob Kurshan (Cadence)
Armin Biere (ETH Zürich)	Yassine Lakhnech (IMAG)
Roderick Bloem (Graz)	Kim Guldstrand Larsen (Aalborg)
Ed Clarke (CMU)	Peter Manolios (Georgia Tech)
Matt Dwyer (Kansas State)	Ken McMillan (Cadence)
E. Allen Emerson (Texas)	Tom Melham (Oxford)
Steven German (IBM)	Chris J. Myers (Utah)
Orna Grumberg (Technion)	Kedar Namjoshi (Bell Labs)
Alan Hu (British Columbia)	Doron A. Peled (Warwick)
Warren A. Hunt, Jr. (Texas, co-chair)	Sriram Rajamani (Microsoft)
Robert B. Jones (Intel)	Thomas Shiple (Synopsys)
Bengt Jonsson (Uppsala)	Fabio Somenzi (Colorado, co-chair)
Nils Klarlund (ATT)	Helmut Veith (TU Wien)
Andreas Kuehlmann (Cadence)	Yaron Wolfsthal (IBM)

## Steering Committee

Edmund M. Clarke (CMU)	Amir Pnueli (Weizmann)
Robert P. Kurshan (Cadence)	Joseph Sifakis (IMAG)

## Sponsors

IBM  
Esterel Technologies  
Intel

## Referees

Mark Aagaard	Gadiel Auerbach	Ritwik Bhattacharya
Parosh Abdulla	Mohammad Awedh	Jesse D. Bingham
Cyril Allauzen	Christel Baier	Per Bjesse
Nina Amla	Thomas Ball	Rastislav Bodik
Torben Amtoft	Jason Baumgartner	Ahmed Bouajjani
Konstantine Arkoudas	Gerd Behrmann	Patricia Bouyer
Cyrille Artho	Shoham Ben-David	Marius Bozga
Eugene Asarin	Johan Bengtsson	Donald Chai
Hossein S. Attar	Sergey Berezin	Sagar Chaki

Ching-Tsun Chou	Charanjit Jutla	Shaz Qadeer
David Dagon	Vineet Kahlon	Ishai Rabinovitz
Anat Dahan	Gila Kamhi	Venkatesh Prasad Ran-
Weizhen Dai	Joost-Pieter Katoen	ganath
Dennis Dams	Sagi Katz	Kavita Ravi
Thao Dang	Shmuel Katz	Sandip Ray
Ashish Darbari	Stefan Katzenbeisser	Erik Reeber
Alexandre David	Sharon Keidar-Barner	Robby
David Deharbe	Mike Kishinevsky	Marco Roveri
William Deng	Daniel Kroening	Sitvanit Ruah
J. Deshmukh	Robert Krug	Hassen Saidi
Rolf Drechsler	Antonin Kucera	Marko Samer
Xiaoqun Du	Jim Kukula	Jun Sawada
Stefan Edelkamp	Ranko Lazic	Karsten Schmidt
Cindy Eisner	Tim Leonard	Viktor Schuppan
Cristian Ene	Flavio Lerda	Assaf Schuster
Kousha Etesami	Martin Leucker	Stefan Schwoon
Eitan Farchi	Bing Li	Bikram Sengupta
Ansgar Fehnker	Yuan Lu	Ohad Shacham
Elena Fersman	Hanbing Lui	Vitaly Shmatikov
Görschwin Fey	Yoad Lustig	Sharon Shoham
Dana Fisman	Tony Ma	Eli Singerman
Emmanuel Fleury	P. Madhusudan	Nishant Sinha
Ruben Gamboa	Monika Maidl	Prasad Sistla
Blaise Genest	Oded Maler	Christian Stangier
Rob Gerth	Freddy Mang	Ofer Strichman
Dimitra Giannakopoulou	S. Manley	Rob Sumners
Alain Girault	Rupak Majumdar	Matyas Sustik
Ganesh Gopalakrishnan	Alexander Medvedev	Bassam Tabbara
Susanne Graf	Todd Millstein	Paulo Tabuada
Alan Hartman	Marius Minea	Murali Talupur
John Hatcliff	Laurent Mounier	Oksana Tkachuk
Klaus Havelund	Anca Muscholl	Richard Treffer
Holger Hermanns	Naren Narasimhan	Stavros Tripakis
Tamir Heyman	Ziv Nevo	Ken Turner
Stefan Höreth	Matthew Nichols	Rachel Tzoref
Neil Immerman	Marcus Nilsson	Serita Van Groningen
Radu Iosif	John O'Leary	Moshe Vardi
S. Iyer	Arlindo Oliveira	Miroslav Velev
Ranjit Jhala	Joel Ouaknine	Bjorn Victor
Wei Jiang	Viresh Paruthi	Catalin Visinescu
HoonSang Jin	Corina Pasareanu	Willem Visser
Rajeev Joshi	Paul Pettersson	Vinod Viswanath
Alma Lizbeth Juarez-	Nir Piterman	Daron Vroon
Dominguez	Mitra Purandare	T. Wahl

VIII      Organization

Chao Wang  
Farn Wang  
Rafael Wisniewski

Jin Yang  
Karen Yorav

Yunshan Zhu  
David Zook



# Table of Contents

## Extending Bounded Model Checking

Interpolation and SAT-Based Model Checking . . . . .	1
<i>K.L. McMillan</i>	

Bounded Model Checking and Induction: From Refutation to Verification . . . . .	14
<i>Leonardo de Moura, Harald Rueß, Maria Sorea</i>	

## Symbolic Model Checking

Reasoning with Temporal Logic on Truncated Paths . . . . .	27
<i>Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, David Van Campenhout</i>	

Structural Symbolic CTL Model Checking of Asynchronous Systems . . . . .	40
<i>Gianfranco Ciardo, Radu Siminiceanu</i>	

A Work-Efficient Distributed Algorithm for Reachability Analysis . . . . .	54
<i>Orna Grumberg, Tamir Heyman, Assaf Schuster</i>	

## Games, Trees, and Counters

Modular Strategies for Infinite Games on Recursive Graphs . . . . .	67
<i>Rajeev Alur, Salvatore La Torre, P. Madhusudan</i>	

Fast Mu-Calculus Model Checking when Tree-Width Is Bounded . . . . .	80
<i>Jan Obdržálek</i>	

Dense Counter Machines and Verification Problems . . . . .	93
<i>Gaoyan Xie, Zhe Dang, Oscar H. Ibarra, Pierluigi San Pietro</i>	

## Tool Presentations I

TRIM: A Tool for Triggered Message Sequence Charts . . . . .	106
<i>Bikram Sengupta, Rance Cleaveland</i>	

Model Checking Multi-Agent Programs with CASP . . . . .	110
<i>Rafael H. Bordini, Michael Fisher, Carmen Pardavila, Willem Visser, Michael Wooldridge</i>	

Monitoring Temporal Rules Combined with Time Series . . . . .	114
<i>Doron Drusinsky</i>	

FAST: Fast Acceleration of Symbolic Transition Systems . . . . .	118
<i>Sébastien Bardin, Alain Finkel, Jérôme Leroux, Laure Petrucci</i>	

Rabbit: A Tool for BDD-Based Verification of Real-Time Systems . . . . .	122
<i>Dirk Beyer, Claus Lewerentz, Andreas Noack</i>	

## Abstraction I

Making Predicate Abstraction Efficient: How to Eliminate Redundant Predicates . . . . .	126
<i>Edmund Clarke, Orna Grumberg, Muralidhar Talupur, Dong Wang</i>	
A Symbolic Approach to Predicate Abstraction . . . . .	141
<i>Shuvendu K. Lahiri, Randal E. Bryant, Byron Cook</i>	
Unbounded, Fully Symbolic Model Checking of Timed Automata Using Boolean Methods . . . . .	154
<i>Sanjit A. Seshia, Randal E. Bryant</i>	

## Dense Time

Digitizing Interval Duration Logic . . . . .	167
<i>Gaurav Chakravorty, Paritosh K. Pandya</i>	
Timed Control with Partial Observability . . . . .	180
<i>Patricia Bouyer, Deepak D'Souza, P. Madhusudan, Antoine Petit</i>	
Hybrid Acceleration Using Real Vector Automata . . . . .	193
<i>Bernard Boigelot, Frédéric Herbreteau, Sébastien Jodogne</i>	

## Tool Presentations II

Abstraction and BDDs Complement SAT-Based BMC in <i>DiVer</i> . . . . .	206
<i>Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, Pranav Ashar</i>	
TLQSolver: A Temporal Logic Query Checker . . . . .	210
<i>Marsha Chechik, Arie Gurfinkel</i>	
Evidence Explorer: A Tool for Exploring Model-Checking Proofs . . . . .	215
<i>Yifei Dong, C.R. Ramakrishnan, Scott A. Smolka</i>	
HERMES: An Automatic Tool for Verification of Secrecy in Security Protocols . . . . .	219
<i>Liana Bozga, Yassine Lakhnech, Michaël Périn</i>	

## Infinite State Systems

Iterating Transducers in the Large . . . . .	223
<i>Bernard Boigelot, Axel Legay, Pierre Wolper</i>	
Algorithmic Improvements in Regular Model Checking . . . . .	236
<i>Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso</i>	

Efficient Image Computation in Infinite State Model Checking . . . . .	249
<i>Constantinos Bartzis, Tevfik Bultan</i>	

## Abstraction II

Thread-Modular Abstraction Refinement . . . . .	262
<i>Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Shaz Qadeer</i>	
A Game-Based Framework for CTL Counterexamples and 3-Valued Abstraction-Refinement . . . . .	275
<i>Sharon Shoham, Orna Grumberg</i>	
Abstraction for Branching Time Properties . . . . .	288
<i>Kedar S. Namjoshi</i>	

## Applications

Certifying Optimality of State Estimation Programs . . . . .	301
<i>Grigore Roşu, Ram Prasad Venkatesan, Jon Whittle, Laurenţiu Leuştean</i>	
Domain-Specific Optimization in Automata Learning . . . . .	315
<i>Hardi Hungar, Oliver Niese, Bernhard Steffen</i>	
Model Checking Conformance with Scenario-Based Specifications . . . . .	328
<i>Marcelo Glusman, Shmuel Katz</i>	

## Theorem Proving

Deductive Verification of Advanced Out-of-Order Microprocessors . . . . .	341
<i>Shuvendu K. Lahiri, Randal E. Bryant</i>	
Theorem Proving Using Lazy Proof Explication . . . . .	355
<i>Cormac Flanagan, Rajeev Joshi, Xinming Ou, James B. Saxe</i>	

## Automata-Based Verification

Enhanced Vacuity Detection in Linear Temporal Logic . . . . .	368
<i>Roy Armoni, Limor Fix, Alon Flaisher, Orna Grumberg, Nir Piterman, Andreas Tiemeyer, Moshe Y. Vardi</i>	
Bridging the Gap between Fair Simulation and Trace Inclusion . . . . .	381
<i>Yonit Kesten, Nir Piterman, Amir Pnueli</i>	
An Improved On-the-Fly Tableau Construction for a Real-Time Temporal Logic . . . . .	394
<i>Marc Geilen</i>	

**Invariants**

Strengthening Invariants by Symbolic Consistency Testing ..... 407  
*Husam Abu-Haimed, Sergey Berezin, David L. Dill*

Linear Invariant Generation Using Non-linear Constraint Solving..... 420  
*Michael A. Colón, Sriram Sankaranarayanan, Henny B. Sipma*

**Explicit Model Checking**

To Store or Not to Store ..... 433  
*Gerd Behrmann, Kim G. Larsen, Radek Pelánek*

Calculating  $\tau$ -Confluence Compositionally ..... 446  
*Gordon J. Pace, Frédéric Lang, Radu Mateescu*

**Author Index** ..... 461

# Interpolation and SAT-Based Model Checking

K.L. McMillan

Cadence Berkeley Labs

**Abstract.** We consider a fully SAT-based method of unbounded symbolic model checking based on computing Craig interpolants. In benchmark studies using a set of large industrial circuit verification instances, this method is greatly more efficient than BDD-based symbolic model checking, and compares favorably to some recent SAT-based model checking methods on positive instances.

## 1 Introduction

Symbolic model checking [8,9] is a method of verifying temporal properties of finite (and sometimes infinite) state systems that relies on a symbolic representation of sets, typically as Binary Decision Diagrams [7] (BDD's). By contrast, bounded model checking [4] can falsify temporal properties by posing the existence of a counterexample of  $k$  steps or fewer as a Boolean satisfiability (SAT) problem. Using a modern SAT solver, this method is efficient in producing counterexamples [10,6]. However, it cannot verify properties unless an upper bound is known on the depth of the state space, which is not generally the case.

This paper presents a purely SAT-based method of *unbounded* model checking. It exploits a SAT solver's ability to produce refutations. In bounded model checking, a refutation is a proof that there is no counterexample of  $k$  steps or fewer. Such a proof implies nothing about the truth of the property in general, but does contain information about the reachable states of the model. In particular, given a partition of a set of clauses into a pair of subsets  $(A, B)$ , and a proof by resolution that the clauses are unsatisfiable, we can generate an *interpolant* in linear time [21]. An interpolant [11] for the pair  $(A, B)$  is a formula  $P$  with the following properties:

- $A$  implies  $P$ ,
- $P \wedge B$  is unsatisfiable, and
- $P$  refers only to the common variables of  $A$  and  $B$ .

Using interpolants, we obtain a complete method for finite-state reachability analysis, and hence LTL model checking, based entirely on SAT.

### 1.1 Related Work

SAT solvers have been applied in unbounded model checking in several ways. For example, they have been used in a hybrid method to detect fixed points, while

the quantifier elimination required for image computations is performed by other means (*e.g.*, by expansion of the quantifier as  $\exists v.f = f\langle 0/v \rangle \vee f\langle 1/v \rangle$ , followed by simplification). Such methods include [5,2,25]. Because of the expense of quantifier elimination, this approach is limited to models with a small number of inputs (typically zero or one). By contrast, the present approach is based entirely on SAT, does not use quantifier elimination, and is not limited in the number of inputs (examples with thousands of inputs have been verified). SAT algorithms have also been used to generate a disjunctive decompositions for BDD-based image computations [13]. Here, BDD's are not used.

Another approach is based on unfolding the transition relation to the length of the longest simple path between two states [22]. The fact that this length has been reached can be verified using a SAT solver. The longest simple path can, however, be exponentially longer than the diameter of the state space (for example, the longest simple path for an  $n$ -bit register is  $2^n$ , while the diameter is 1). The present method does not require unfolding beyond the diameter of the state space, and in practice often succeeds with shorter unfoldings.

Finally, Baumgartner, *et al.* [3], use SAT-based bounded model checking with a structural method for bounding the depth of the state space. This requires the circuit in question to have special structure and does not always give useful bounds. In a suite of benchmarks, we find that the present method successfully resolves almost all of the model checking problems that could not be resolved by the structural method.

## 1.2 Outline

The next section covers resolution proofs and interpolation. Then in section 4 we give a method for unbounded model checking based on interpolation. Finally, in section 5, we test the method in practice, applying it to the verification of some properties of commercial microprocessor designs.

## 2 Interpolation Algorithm

To begin at the beginning, a *clause* is a disjunction of zero or more *literals*, each being either a Boolean variable or its negation. We assume that clauses are *non-tautological*, that is, no clause contains a variable and its negation. A clause set is *satisfiable* when there is a truth assignment to the Boolean variables that makes all clauses in the set true.

Given two clauses of the form  $c_1 = v \vee A$  and  $c_2 = \neg v \vee B$ , we say that the *resolvent* of  $c_1$  and  $c_2$  is the clause  $A \vee B$ , provided  $A \vee B$  is non-tautological. For example, the resolvent of  $a \vee b$  and  $\neg a \vee \neg c$  is  $b \vee \neg c$ , while  $a \vee b$  and  $\neg a \vee \neg b$  have no resolvent, since  $b \vee \neg b$  is tautological. It is easy to see that any two clauses have at most one resolvent. The resolvent of  $c_1$  and  $c_2$  (if it exists) is a clause that is implied by  $c_1 \wedge c_2$  (in fact, it is exactly  $(\exists v)(c_1 \wedge c_2)$ ). We will call  $v$  the *pivot variable* of  $c_1$  and  $c_2$ .

**Definition 1.** A proof of unsatisfiability  $\Pi$  for a set of clauses  $C$  is a directed acyclic graph  $(V_\Pi, E_\Pi)$ , where  $V_\Pi$  is a set of clauses, such that

- for every vertex  $c \in V_\Pi$ , either
  - $c \in C$ , and  $c$  is a root, or
  - $c$  has exactly two predecessors,  $c_1$  and  $c_2$ , such that  $c$  is the resolvent of  $c_1$  and  $c_2$ , and
- the empty clause is the unique leaf.

**Theorem 1.** If there is a proof of unsatisfiability for clause set  $C$ , then  $C$  is unsatisfiable.

A SAT solver, such as CHAFF [18], or GRASP [23], is a complete decision procedure for clause sets. In the satisfiable case, it produces a satisfying assignment. In the unsatisfiable case, it can produce a proof of unsatisfiability [17,26]. This, in turn, can be used to generate an interpolant by a very simple procedure [21]. This procedure produces a Boolean circuit whose gates correspond to the vertices (*i.e.*, resolution steps) in the proof. The procedure given here is similar but not identical to that in [21].

Suppose we are given a pair of clause sets  $(A, B)$  and a proof of unsatisfiability  $\Pi$  of  $A \cup B$ . With respect to  $(A, B)$ , say that a variable is *global* if it appears in both  $A$  and  $B$ , and *local* to  $A$  if it appears only in  $A$ . Similarly, a literal is global or local to  $A$  depending on the variable it contains. Given any clause  $c$ , we denote by  $g(c)$  the disjunction of the global literals in  $c$  and by  $l(c)$  the disjunction literals local to  $A$ .

For example, suppose we have two clauses,  $c_1 = (a \vee b \vee \neg c)$  and  $c_2 = (b \vee c \vee \neg d)$ , and suppose that  $A = \{c_1\}$  and  $B = \{c_2\}$ . Then  $g(c_1) = (b \vee \neg c)$ ,  $l(c_1) = (a)$ ,  $g(c_2) = (b \vee c)$  and  $l(c_2) = \text{FALSE}$ .

**Definition 2.** Let  $(A, B)$  be a pair of clause sets and let  $\Pi$  be a proof of unsatisfiability of  $A \cup B$ , with leaf vertex  $\text{FALSE}$ . For all vertices  $c \in V_\Pi$ , let  $p(c)$  be a boolean formula, such that

- if  $c$  is a root, then
  - if  $c \in A$  then  $p(c) = g(c)$ ,
  - else  $p(c)$  is the constant  $\text{TRUE}$ .
- else, let  $c_1, c_2$  be the predecessors of  $c$  and let  $v$  be their pivot variable:
  - if  $v$  is local to  $A$ , then  $p(c) = p(c_1) \vee p(c_2)$ ,
  - else  $p(c) = p(c_1) \wedge p(c_2)$ .

The  $\Pi$ -interpolant of  $(A, B)$ , denoted  $\text{ITP}(\Pi, A, B)$  is  $p(\text{FALSE})$ .

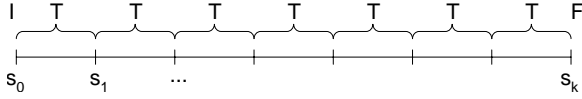
**Theorem 2.** For all  $(A, B)$ , a pair of clause sets, and  $\Pi$ , a proof of unsatisfiability of  $A \cup B$ ,  $\text{ITP}(\Pi, A, B)$  is an interpolant for  $(A, B)$ .

The formula  $\text{ITP}(\Pi, A, B)$  can be computed in time  $O(N + L)$ , where  $N$  is the number of vertices in the proof  $|V_\Pi|$  and  $L$  is the total number of literals in the proof  $\sum_{c \in V_\Pi} |c|$ . Its circuit size is also  $O(N + L)$ . Of course, the size of the proof  $\Pi$  is exponential in the size of  $A \cup B$  in the worst case.

### 3 Model Checking Based on Interpolation

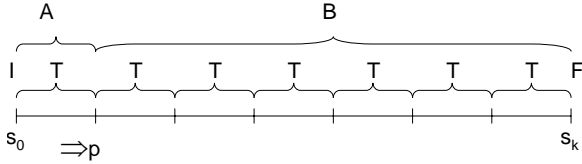
Bounded model checking and interpolation can be combined to produce an over-approximate image operator that can be used in symbolic model checking.

The intuition behind this is as follows. A bounded model checking problem consists of a set of constraints – initial constraints, transition constraints, final constraints. These constraints are translated to conjunctive normal form, and, as appropriate, instantiated for each time frame  $0 \dots k$ , as depicted in Figure 1. In the figure,  $I$  represents the initial constraint,  $T$  the transition constraint, and  $F$  the final constraint. Now suppose that we partition the clauses so that the



**Fig. 1.** Bounded model checking.

initial constraint and first instance of the transition constraint are in set  $A$ , while the final condition and the remaining instances of the transition constraint are in set  $B$ , as depicted in Figure 2. The common variables of  $A$  and  $B$  are exactly the variables representing state  $s_1$ .



**Fig. 2.** Computing image by interpolation.

Using a SAT solver, we prove the clause set is unsatisfiable (i.e., there are no counterexamples of length  $k$ ). From the proof we derive an interpolant  $P$  for  $(A, B)$ . Since  $P$  is implied by the initial condition and the first transition constraint, it follows that  $P$  is true in every state reachable from the initial state in one step. That is,  $P$  is an over-approximation of the forward image of  $I$ . Further,  $P$  and  $B$  are unsatisfiable, meaning that no state satisfying  $P$  can reach a final state in  $k - 1$  steps.

This over-approximate image operation can be iterated to compute an over-approximation of the reachable states. Because of the approximation, we may falsely conclude that  $F$  is reachable. However, by increasing  $k$ , we must eventually find a true counterexample (a path from  $I$  to  $F$ ) or prove that  $F$  is not reachable (i.e., the property is true), as we shall see.



### 3.1 Basic Model Checking Algorithm

The LTL model checking problem can be reduced to finding an accepting run of a finite automaton. This translation has been extensively studied [19,24,14], and will not be described here. Moreover, we need consider only the problem of finding finite counterexamples to safety properties. Liveness properties can then be handled by the method of [1]. We assume that the problem of safety property verification is posed in terms of a one-letter automaton on finite words, such that the property is false exactly when the automaton has an accepting run. Such a construction can be found, for example, in [15].

The automaton itself will be represented implicitly by Boolean formulas. The state space of the automaton is defined by an indexed set of Boolean variables  $V = \{v_1, \dots, v_n\}$ . A *state*  $S$  is a corresponding vector  $(s_1, \dots, s_n)$  of Boolean values. A *state predicate*  $P$  is a Boolean formula over  $V$ . We will write  $P(W)$  to denote  $P\langle w_i/v_i \rangle$  (that is,  $p$  with  $w_i$  substituted for each  $v_i$ ). We also assume an indexed set of “next state” variables  $V' = \{v'_1, \dots, v'_n\}$ , disjoint from  $V$ . A *state relation*  $R$  is a Boolean formula over  $V$  and  $V'$ . We will write  $R(W, W')$  to denote  $R\langle w_i/v_i, w'_i/v'_i \rangle$ .

For our purposes, an *automaton* is a triple  $M = (I, T, F)$ , where the initial constraint  $I$  and final constraint  $F$  are state predicates, and the transition constraint  $T$  is a state relation. A *run* of  $M$ , of length  $k$ , is a sequence of states  $s_0 \dots s_k$  such that  $I(s_0)$  is true, and for all  $0 \leq i < k$ ,  $T(s_i, s_{i+1})$  is true, and  $F(s_k)$  is true. In bounded model checking, we would translate the existence of a run of length  $j \leq i \leq k$  into a Boolean satisfiability problem by introducing a new indexed set of variables  $W_i = \{w_{i1}, \dots, w_{in}\}$ , for  $0 \leq i \leq k$ . A run of length in the range  $j \dots k$  exists exactly when the following formula is satisfiable:<sup>1</sup>

$$\text{BMC}_j^k = I(W_0) \wedge \left( \bigwedge_{0 \leq i < k} T(W_i, W_{i+1}) \right) \wedge \left( \bigvee_{j \leq i \leq k} F(W_i) \right)$$

We will divide this formula into two parts: one formula representing the possible prefixes of a run, and another representing the possible suffixes. The possible prefixes of length  $l$  are characterized by the following formula:

$$\text{Pref}_l(M) = I(W_{-l}) \wedge \left( \bigwedge_{-l \leq i < 0} T(W_i, W_{i+1}) \right)$$

That is, a prefix begins in an initial state  $W_{-l}$  and ends in any state  $W_0$ . The possible suffixes of length  $j \dots k$  are characterized by the following formula:

$$\text{Suff}_j^k(M) = \left( \bigwedge_{0 \leq i < k} T(W_i, W_{i+1}) \right) \wedge \left( \bigvee_{j \leq i \leq k} F(W_i) \right)$$

---

<sup>1</sup> Actually, this characterization is correct only if transition relation is total. In this paper we will assume that transition relations are total by construction. The generalization to partial transition relations is not difficult, however.

A suffix begins in any state  $W_0$  and ends in some final state  $W_i$ , where  $j \leq i \leq k$ .

To apply a SAT solver, we must translate Boolean formulas into conjunctive normal form. Here, we simply assume the existence of some function  $\text{CNF}$  that translates a Boolean formula  $f$  into a set of clauses  $\text{CNF}(f, U)$ , where  $U$  is a set of “fresh” variables, not occurring in  $f$ . The translation function  $\text{CNF}$  must have the property that  $(\exists U. \text{CNF}(f, U)) \equiv f$ . That is, the satisfying assignments of  $\text{CNF}(f, U)$  are exactly those of  $f$ , if we ignore the fresh variables. A suitable translation that is linear in the formula size can be found in [20]. What follows, however, does not depend on the precise translation function.

A procedure to check the existence of a finite run of  $M$  is shown in Figure 3. In the figure,  $U_1$  and  $U_2$  are assumed to be sets of fresh variables, disjoint from each other and all the  $W_i$ ’s. The procedure is parameterized by a fixed value  $k \geq 0$ . We will show that the procedure must terminate for sufficiently large values of  $k$ , though for small values it may abort, without deciding the existence of a run. The procedure runs as follows. First, we check that there is no run of length zero. Assuming there is not, we set our initial approximation  $R$  of the reachable states to be  $I$ , the initial states. We then compute an over-approximation of the forward image of  $R$ . This is done by treating  $R$  as the initial condition and checking the satisfiability of the formula  $\text{PREF}_1(M) \wedge \text{SUFF}_0^k(M)$ . If this is satisfiable there is a run of length  $1 \dots k + 1$ , starting at  $R$  and ending at  $F$ . In the first iteration, when  $R = I$ , we have found a run of the automaton, and we terminate. If  $R \neq I$ , we abort, without deciding the existence of a run.

On the other hand, suppose that  $\text{PREF}_1(M) \wedge \text{SUFF}_0^k(M)$  is unsatisfiable. Using the proof of unsatisfiability  $\Pi$ , we construct a  $\Pi$ -interpolant  $P$  for the pair  $(\text{PREF}_1(M), \text{SUFF}_0^k(M))$ . Since  $P$  is a formula that is implied by  $R(W_{-1})$  and  $T(W_{-1}, W_0)$ , we know that  $P$  holds in all states  $W_0$  reachable from  $R$  in one step (or put another way, it is an *over-approximation* of the states reachable in one step). Further, since  $P$  and  $\text{SUFF}_0^k(M)$  are inconsistent, no state satisfying  $P$  can reach  $F$  in up to  $k$  steps (that is,  $P$  is an *under-approximation* of the states that are backward reachable from  $F$  in up to  $k$  steps). Thus, we obtain a new approximation  $R \vee P(V/W_0)$  of the reachable states. If a fixed point is reached,  $R$  is an inductive invariant. Since no state in  $R$  satisfies  $F$  (nor can reach  $F$  in up to  $k$  steps), we terminate, indicating that no run exists. Otherwise, we continue the procedure with the new value of  $R$ .

**Theorem 3.** *For  $k > 0$ , if  $\text{FINITERUN}(M, k)$  terminates without aborting, it returns TRUE iff  $M$  has a run.*

**Proof.** Suppose the procedure returns TRUE. Either  $I \wedge F$  is satisfiable, in which case  $M$  has a run of length 0, or  $\text{BMC}_1^k(M)$  is satisfiable, hence  $M$  has a run of length  $1 \dots k$ . Now, suppose the procedure returns FALSE. We can show:

1.  $I$  implies  $R$  (trivial).
2.  $R$  is an invariant of  $T$  (in other words,  $R(s)$  and  $T(s, s')$  imply  $R(s')$ ). Since  $\text{PREF}_1(M')$  implies  $P$ , it follows that, for all states  $s, s'$ ,  $R(s) \wedge T(s, s') \Rightarrow R'(s')$ . Thus, when  $R'$  implies  $R$ , we have  $R(s)$  and  $T(s, s')$  implies  $R(s')$ .

```

procedure FINITERUN( $M = (I, T, F)$ ,  $k > 0$ )
  if  $I \wedge F$  is satisfiable, return TRUE
  let  $R = I$ 
  while true
    let  $M' = (R, T, F)$ 
    let  $A = \text{CNF}(\text{PREF}_1(M'), U_1)$ 
    let  $B = \text{CNF}(\text{SUFF}_0^k(M'), U_2)$ 
    Run SAT on  $A \cup B$ . If satisfiable, then
      if  $R = I$  return TRUE else abort
    else (if  $A \cup B$  unsatisfiable)
      let  $\Pi$  be a proof of unsatisfiability of  $A \cup B$ 
      let  $P = \text{ITP}(\Pi, A, B)$ 
      let  $R' = P\langle W/W_0 \rangle$ .
      if  $R'$  implies  $R$  return FALSE
      let  $R = R \vee R'$ 
end

```

**Fig. 3.** Procedure for existence of a finite run

3.  $R \wedge F$  is unsatisfiable. Initially,  $R = I$  and  $I \wedge F$  is unsatisfiable. At each iteration, we know  $P \wedge \text{SUFF}_0^k(M')$  is unsatisfiable, hence  $R' \wedge F$  is unsatisfiable (assuming  $T$  is total).

It follows by induction that  $M$  has no run of any length.  $\square$

We can also show that the procedure must terminate for sufficiently large values of  $k$ . Let us define the *reverse depth* of  $M$  as the maximum length of the shortest path from any state to a state satisfying  $F$ . This can also be viewed as the depth of a breadth-first backward traversal from  $F$ . This depth is bounded by  $2^{|V|}$  but in most practical cases is much smaller.

**Theorem 4.** *For every  $M$ , there exists  $k$  such that  $\text{FINITERUN}(M, k)$  terminates.*

**Proof.** Let  $k$  be the reverse depth of  $M$ . In the first iteration, if the SAT problem is satisfiable, the procedure terminates. Otherwise,  $R'$  cannot reach  $F$  in  $k$  steps. Since  $k$  is the reverse depth, it follows that  $R'$  cannot reach  $F$  in any number of steps. Thus, at the next iteration  $R$  cannot reach  $F$  in  $k + 1$  steps, so the SAT problem must again be unsatisfiable. Carrying on by induction, we conclude that at every iteration,  $R$  cannot reach  $F$  in up to  $k + 1$  steps. Thus  $R$  must continue to increase (*i.e.*, become weaker) until it reaches a fixed point, at which time the procedure terminates.  $\square$

Thus, when procedure  $\text{FINITERUN}$  aborts, we have only to increase the value of  $k$ . If we continue to do this, eventually  $\text{FINITERUN}$  will terminate. The amount by which to increase  $k$  has some bearing on performance. If we increase it by too little, we waste time on aborted runs. If we increase it by too much the resulting SAT problems may become intractable. In practice,  $\text{FINITERUN}$  often terminates for values of  $k$  substantially smaller than the reverse depth.

### 3.2 Optimizations

The basic algorithm can be improved in several ways. First, the interpolants are typically highly redundant, in that many subformulas are syntactically distinct but logically equivalent. Eliminating redundant subformulas thus greatly reduces the size of the interpolant. There is a large literature on identifying logically equivalent formulas. For this paper, a simple method of building BDD's up to a small fixed size was used.

Second, we can replace  $\text{SUFF}_0^k$  with  $\text{SUFF}_j^k$ , for some  $j > 0$  (*i.e.*, we test the property for times greater than or equal to  $j$ ). In most cases, setting  $j = k$  to produces the best results, since the SAT solver only needs to refute the final condition at a single step, rather than at all steps. Unfortunately, if  $j > 0$ , there is no guarantee of termination, except when the runs of the automaton are stuttering closed. In practice divergence has been observed for a few hardware models with two-phase clocks. This was correctable by setting  $j = k - 1$ . Clearly, some automated means is needed to set the value of  $j$ , but as yet this has not been developed.

Third, we can use “frontier set simplification”. That is, it suffices to compute the forward image approximation of the “new” states, rather than the entire reachable set  $R$ . In fact, any set intermediate between these will do. Since we use arbitrary Boolean formulas rather than BDD's, there is no efficient method available for this simplification. In this work, we simply use  $R'$  (the previous image result) in place of  $R$ .

Finally, note that the formula  $\text{SUFF}_j^k(M')$  is invariant from one iteration of the next. It constitutes most of the CNF formula that the SAT solver must refute. Clearly it is inefficient to rebuild this formula at each iteration. A better approach would be to keep all the clauses of  $\text{SUFF}_j^k(M')$ , and all the clauses inferred from these, from one run of the SAT solver to the next. That was not done here because it would require modification of the SAT solver. However, the potential savings in run time is substantial.

## 4 Practical Experience

The performance of the interpolation-based model checking procedure was tested on two sets of benchmark problems derived from commercial microprocessor designs. The first is a sampling of properties from the compositional verification of a unit of the Sun PicoJava II microprocessor.<sup>2</sup> This unit is the ICU, which manages the instruction cache, prefetches instructions, and does partial instruction decoding. Originally, the properties were verified by standard symbolic model checking, using manual directives to remove irrelevant parts of the logic. To make difficult benchmark examples, these directives were removed, and a neighboring unit, the instruction folding unit (IFU), was added. The IFU reads instruction

---

<sup>2</sup> The tools needed to construct the benchmark examples from the PicoJava II source code can be found at <http://www-cad.eecs.berkeley.edu/~kemmcmil>.

bytes from the instruction queue, parses the byte stream into separate instructions and divides the instructions into groups that can be executed in a single cycle. Inclusion of the IFU increases the number of state variables in the “cone of influence” substantially, largely by introducing dependencies on registers within the ICU itself. It also introduces a large amount of irrelevant combinational logic.

Twenty representative properties were chosen as benchmarks. All are safety properties, of the form  $Gp$ , where  $p$  is a formula involving only current time and the next time (usually only current time). The number of state variables in these problems after the cone of influence reduction ranges from around 50 to around 350. All the properties are true. Tests were performed on a Linux workstation with a 930MHz Pentium III processor and 512MB of available memory. Unbounded BDD-based symbolic model checking was performed using the Cadence SMV system. SAT solving was performed using an implementation of the BerkMin algorithm [12], modified to produce proofs of unsatisfiability.

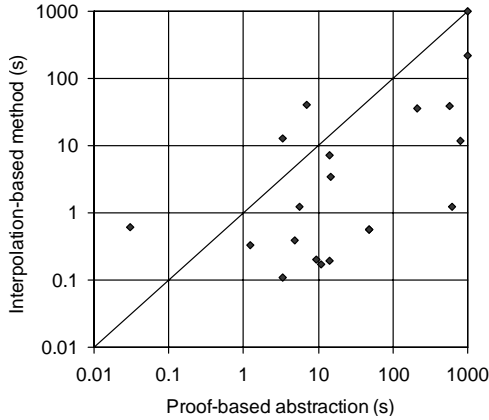
No property could be verified by standard symbolic model checking, within a limit of 1800 seconds. On the other hand, of the 20 properties, 19 were successfully verified by the interpolation method.

Figure 4 shows a comparison of the interpolation method against another method called *proof-based abstraction* that uses SAT to generate abstractions which are then verified by standard symbolic model checking [17]. This method is more effective than simple BDD-based model checking, successfully verifying 18 of the 20 properties. In the figure, each point represents one benchmark problem, the X axis representing the time in seconds taken by the proof-based abstraction method, and the Y axis representing the time in seconds taken by the interpolation method.<sup>3</sup> A time value of 1000 indicates a time-out after 1000 seconds. Points below the diagonal indicate an advantage for the present method. We observe 16 wins for interpolation and 3 for proof-based abstraction, with one problem solved by neither method. In five or six cases, the interpolation method wins by two orders of magnitude.

Figure 5 compares the performance of the interpolation approach with results previously obtained by Baumgartner *et al.* [3] on a set of model checking problems derived from the IBM Gigahertz Processor. Their method involved a combination of SAT-based bounded model checking, structural methods for bounding the depth of the state space, and target enlargement using BDD's. Each point on the graph represents the average verification or falsification time for a collection of properties of the same circuit model. The average time reported by Baumgartner *et al.* is on the X axis, while the average time for the present method is on the Y axis.<sup>4</sup> A point below the diagonal line represents a lower average time for the interpolation method for one benchmark set. We

<sup>3</sup> Times for the interpolation method include only the time actually used by the SAT solver. Overhead in generating the unfoldings is not counted, since this was implemented inefficiently. An efficient implementation would re-use the unfolding from one iteration to the next, thus making the unfolding overhead negligible. Time to generate the interpolants was negligible. A value of  $j = k$  was used for these runs.

<sup>4</sup> The processor speeds for the two sets of experiments are slightly different. Baumgartner *et al.* used an 800MHz Pentium III, as compared to a 930 MHz Pentium III



**Fig. 4.** Run times on PicoJava II benchmarks.

note 21 wins for interpolation and 3 for the structural method. In a few cases the interpolation method wins by two orders of magnitude. A time of 1000 seconds indicates that the truth of one or more properties in the benchmark could not be determined (either because of time-out or incompleteness). Of the 28 individual properties that could not be resolved by Baumgartner *et al.*, all but one are successfully resolved by the proof partition method.

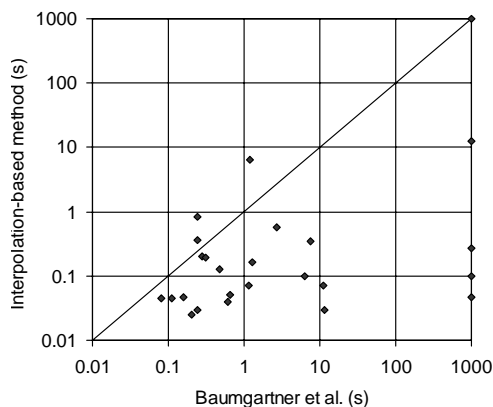
Finally, we compare the interpolation method against proof-based abstraction on the IBM benchmarks. The results are shown in Figure 6. Though the results are mixed, we find that overall the advantage goes to proof-based abstraction (both successfully solve the same set of problems). This appears to be due to the fact that a large number of properties in the benchmark set are false (*i.e.*, have counterexamples). The proof-based abstraction method tends to find counterexamples more quickly because in effect the BDD-based model checker quickly guides the bounded model checker to the right depth, while the interpolation method systematically explores all depths. Figure 7 compares the two methods on only those individual properties that are true, showing an advantage for interpolation. This suggests that a hybrid method might provide the best results overall.

## 5 Conclusion

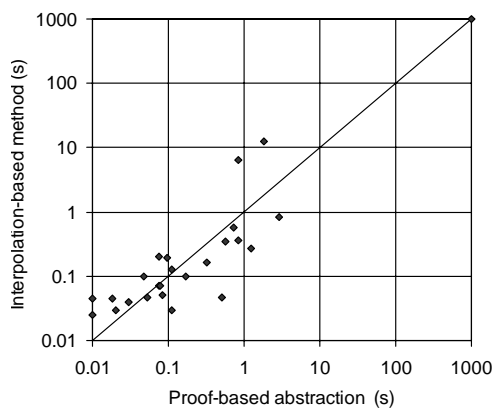
We have observed that interpolation and bounded model checking can be combined to allow unbounded model checking. This method was seen in two micro-processor verification benchmark studies to be more efficient than BDD-based model checking and some recently developed SAT-based methods, for true properties.

---

used here. No adjustment has been made for CPU speed. A value of  $j = k - 1$  was used for these runs, since one problem was found to diverge for  $j = k$ .



**Fig. 5.** Run times on IBM Gigahertz Processor benchmarks.



**Fig. 6.** Run times on IBM Gigahertz Processor benchmarks.

For future work, it is interesting to consider what other information can be extracted from proofs of unsatisfiability that might be useful in model checking. For example, it is possible to derive an abstraction of the transition relation from a bounded model checking refutation, using interpolation. Initial studies have shown that this abstraction is difficult to handle with current BDD-based model checking methods, which rely on a certain structure in the transition relation formula. If this difficulty can be overcome, however, it might lead to an improvement in the proof-based abstraction method. It is also conceivable that interpolation in first-order theories could be applied in infinite-state model checking.

**Acknowledgment** The author would like to thank Jason Baumgartner of IBM for providing the Gigahertz Processor benchmark problems.

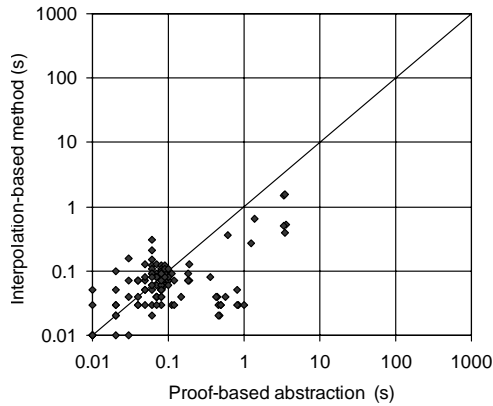


Fig. 7. Run times on IBM Gigahertz Processor true properties.

## References

1. C. Artho A. Biere and V. Schuppan. Liveness checking as safety checking. In *Formal Methods for Industrial Critical Systems (FMICS'02)*, July 2002.
2. P.A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.
3. J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification (CAV 2002)*, pages 151–165, 2002.
4. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*, pages 193–207, 1999.
5. P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-100, Department of Computer Science, Chalmers technical university, March 1999.
6. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification (CAV 2001)*, 2001.
7. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
8. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
9. O.C., C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
10. F. Copt, L. Fix, Fraer R, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking in an industrial setting. In *Computer Aided Verification (CAV 2001)*, pages 436–453, 2001.
11. W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250–268, 1957.
12. E. Goldberg and Y. Novikov. BerkMin: a fast and robust SAT-solver. In *DATE 2002*, pages 142–149, 2002.



13. A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *FMCAD 2000*, pages 354–371, 2000.
14. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking:  $10^{20}$  States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
15. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. K.L. McMillan and Nina Amla. Automatic abstraction without counterexamples. To appear, TACAS’03.
17. K.L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS’03*, pages 2–17, 2003.
18. M.W. Moskewicz, C.F. Madigan, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.
19. A. Pnueli O. Lichtenstein. Checking that finite state concurrent programs satisfy their linear specification. In *Principles of Programming Languages (POPL ’85)*, pages 97–107, 1985.
20. D. Plaisted and S. Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
21. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(2):981–998, June 1997.
22. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.
23. J.P.M. Silva and K.A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design, November 1996*, 1996.
24. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science (LICS ’86)*, pages 322–331, 1986.
25. P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.
26. L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE’03*, pages 880–885, 2003.

# Bounded Model Checking and Induction: From Refutation to Verification \*

(Extended Abstract, Category A)

Leonardo de Moura, Harald Rueß, and Maria Sorea\*\*

SRI International  
Computer Science Laboratory  
333 Ravenswood Avenue  
Menlo Park, CA 94025, USA  
{demoura, rueß, sorea}@csl.sri.com  
<http://www.csl.sri.com/>

**Abstract.** We explore the combination of bounded model checking and induction for proving safety properties of infinite-state systems. In particular, we define a general  $k$ -induction scheme and prove completeness thereof. A main characteristic of our methodology is that strengthened invariants are generated from failed  $k$ -induction proofs. This strengthening step requires quantifier-elimination, and we propose a *lazy* quantifier-elimination procedure, which delays expensive computations of disjunctive normal forms when possible. The effectiveness of induction based on bounded model checking and invariant strengthening is demonstrated using infinite-state systems ranging from communication protocols to timed automata and (linear) hybrid automata.

## 1 Introduction

Bounded model checking (BMC) [5,4,7] is often used for refutation, where one systematically searches for counterexamples whose length is bounded by some integer  $k$ . The bound  $k$  is increased until a bug is found, or some pre-computed *completeness threshold* is reached. Unfortunately, the computation of completeness thresholds is usually prohibitively expensive and these thresholds may be too large to effectively explore the associated bounded search space. In addition, such completeness thresholds do not exist for many infinite-state systems.

In deductive approaches to verification, the *invariance rule* is used for establishing invariance properties  $\varphi$  [11,10,13,3]. This rule requires a property  $\psi$  which is stronger than  $\varphi$  and *inductive* in the sense that all initial states satisfy  $\psi$ , and  $\psi$  is preserved under each transition. Theoretically, the invariance rule is adequate for verifying a valid property of a system, but its application usually

---

\* Funded by SRI International, by NSF Grant CCR-0082560, DARPA/AFRL-WPAFB Contract F33615-01-C-1908, and NASA Contract B09060051.

\*\* Also affiliated with University of Ulm, Germany.

requires creativity in coming up with a sufficiently strong inductive invariant. It is also nontrivial to detect bugs from failed induction proofs.

In this paper, we explore the combination of BMC and induction based on the *k-induction* rule. This induction rule generalizes BMC in that it requires demonstrating the invariance of  $\varphi$  in the first  $k$  states of any execution. Consequently, error traces of length  $k$  are detected. This induction rule also generalizes the usual invariance rule in that it requires showing that if  $\varphi$  holds in every state of every execution of length  $k$ , then every successor state also satisfies  $\varphi$ . In its pure form, however, *k-induction* does not require the invention of a strengthened inductive invariant. As in BMC, the bound  $k$  is increased until either a violation is detected in the first  $k$  states of an execution or the property at hand is shown to be *k-inductive*. In the ideal case of attempting to prove correctness of an inductive property, 1-induction suffices and iteration up to a, possibly large, complete threshold, as in BMC, is avoided. The *k-induction* rule is sound, but further conditions, such as the restriction to acyclic execution sequences, must be added to make *k-induction* complete even for finite-state systems [17].

One of our main contributions is the definition of a general *k-induction* rule and a corresponding completeness result. This induction rule is parameterized with respect to suitable notions of simulation. These simulation relations induce different notions of path *compression* in that an execution path is compressed if it does not contain two similar states. Many completeness results, such as *k-induction* for timed automata, follow by simply instantiating this general result with the simulation relation at hand. For general transition systems, we develop an *anytime* algorithm for approximating adequate simulation relations for *k-induction*.

Whenever *k-induction* fails to prove a property  $\varphi$ , there is a counterexample of length  $k + 1$  such that the first  $k$  states satisfy  $\varphi$  and the last state does not satisfy  $\varphi$ . If the first state of this trace is reachable, then  $\varphi$  is refuted. Otherwise, the counterexample is labeled *spurious*. By assuming the first state of this trace is unreachable, a spurious counterexample is used to automatically obtain a strengthened invariant. Many infinite-state systems can only be proven with *k-induction* enriched with invariant strengthening, whereas for finite systems the use of strengthening decreases the minimal  $k$  for which a *k-induction* proof succeeds.

Since our invariant strengthening procedure for *k-induction* heavily relies on eliminating existentially quantified state variables, we develop an effective quantifier elimination algorithm for this purpose. The main characteristic of this algorithm is that it avoids a potential exponential blowup in the initial computation of a disjunctive normal form whenever possible, and a constraint solver is used to identify relevant conjunctions. In this way the paradigm of *lazy* theorem proving, as developed by the authors for the ground case [7], is extended to first-order formulas.

The paper is organized as follows. Section 2 contains background material on encodings of transition systems in terms of logic formulas. In Section 3 we develop the notions of reverse and direct simulations together with an anytime

algorithm for computing these relations. Reverse and direct simulations are used in Section 4 to state a generic  $k$ -induction principle and to provide sufficient conditions for the completeness of these inductions. Sections 5 and 6 discuss invariant strengthening and lazy quantifier elimination. Experimental results with  $k$ -induction and invariant strengthening for various infinite-state protocols, timed automata, and linear hybrid systems are summarized in Section 7. Comparisons to related work are in Section 8.

## 2 Background

Let  $V := \{x_1, \dots, x_n\}$  be a set of variables interpreted over nonempty domains  $\mathcal{D}_1$  through  $\mathcal{D}_n$ , together with a type assignment  $\tau$  such that  $\tau(x_i) = \mathcal{D}_i$ . For a set of typed variables  $V$ , a *variable assignment* is a function  $\nu$  from variables  $x \in V$  to an element of  $\tau(x)$ . The variables in  $V := \{x_1, \dots, x_n\}$  are also called *state variables*, and a *program state* is a variable assignment over  $V$ .

All the developments in this paper are parametric with respect to a given constraint theories  $\mathcal{C}$ , such as linear arithmetic or a theory of bitvectors. We assume a computable function for deciding satisfiability of a conjunction of constraints in  $\mathcal{C}$ . A set of *Boolean constraints*,  $\text{Bool}(\mathcal{C})$ , includes all constraints in  $\mathcal{C}$  and is closed under conjunction  $\wedge$ , disjunction  $\vee$ , and negation  $\neg$ . Effective solvers for deciding the satisfiability problem in  $\text{Bool}(\mathcal{C})$  have been previously described [7,6].

A tuple  $\langle V, I, T \rangle$  is a  $\mathcal{C}$ -*program* over  $V$ , where interpretations of the typed variables  $V$  describe the set of states,  $I \in \text{Bool}(\mathcal{C}(V))$  is a predicate that describes the initial states, and  $T \in \text{Bool}(\mathcal{C}(V \cup V'))$  specifies the transition relation between current states and their successor states ( $V$  denotes the current state variables, while  $V'$  stands for the next state variables). The semantics of a program is given in terms of a *transition system*  $M$  in the usual way.

For a program  $M = \langle V, I, T \rangle$ , a sequence of states  $\pi(s_0, s_1, \dots, s_n)$  forms a *path* through  $M$  if  $\bigwedge_{0 \leq i < n} T(s_i, s_{i+1})$ . A state  $s$  is *reachable* in  $M$  if there is a path  $\pi(s_0, s_1, \dots, s_{n-1}, s)$  through  $M$  and  $I(s_0)$ , and a state property  $\varphi \in \mathcal{C}(V)$  is *invariant* in  $M$  iff  $\varphi(s)$  holds for every reachable state  $s$  in  $M$ . A *counterexample* for a property  $\varphi$  is a path  $\pi(s_0, \dots, s_n)$  such that  $I(s_0)$  and  $\neg\varphi(s_n)$ , and the length  $\text{len}(\pi)$  of such a counterexample is given by the number of states in this path.

Typical programming constructs can be rewritten into the program syntax presented above. For example, Dijkstra's guarded commands are encoded in terms of a disjunction of conjunctions of guards  $g(x_1, \dots, x_n)$  and updates  $x'_i = f_1(x_1, \dots, x_n)$  for all variables  $x_i$ . Programs with external, non-deterministic inputs are defined by partitioning the set of variables into input variables, which are unconstrained, and the other state variables, whose next-state values are constrained by the transition relation.

Throughout this paper we use timed automata [2], which are state-transition graphs augmented with a finite set of real-valued clocks, as a prototypical class of infinite-state systems. Decidability of the model-checking problem for timed

automata rests on the fact that the space of clock valuations is partitioned into finitely many clock regions. Two clock valuations  $v_1, v_2$  that belong to the same region are (region) equivalent, denoted as  $v_1 \sim_{TA} v_2$ . This region equivalence is a *stable* quotient relation, that is, whenever  $q \sim_{TA} u$  and  $T(q, q')$ , there exists a state  $u'$  such that  $T(u, u')$  and  $q' \sim_{TA} u'$  [2]. Encoding of timed automata in terms of logical programs with linear arithmetic constraints are described in [19]. In particular, program states consist of a location and nonnegative real interpretations of clocks. For timed automata we restrict ourselves to proving so-called clock constraints  $\varphi$ , such that  $q \sim_{TA} u$  implies that  $\varphi(q)$  iff  $\varphi(u)$ .

### 3 Direct and Reverse Simulation

The notions of direct and reverse simulation as developed here lay out the foundation for the completeness results in Section 4.

**Definition 1 (Direct / Reverse Simulation).** Let  $M = \langle V, I, T \rangle$  be a program and  $\varphi$  a state formula over  $V$ . We define the functors  $F_d$  and  $F_r$  that map binary relations  $R$  over  $V$  in the following way.

$$F_d(R)(s_1, s_2) := \begin{cases} \text{if } \neg\varphi(s_1) \text{ then } \neg\varphi(s_2) \\ \text{else } \forall s'_1 . T(s_1, s'_1) \Rightarrow \exists s'_2 . R(s'_1, s'_2) \wedge T(s_2, s'_2) \end{cases}$$

$$F_r(R)(s_1, s_2) := \begin{cases} \text{if } I(s_1) \text{ then } I(s_2) \\ \text{else } \forall s'_1 . T(s'_1, s_1) \Rightarrow \exists s'_2 . R(s'_1, s'_2) \wedge T(s'_2, s_2) \end{cases}$$

A *direct simulation* over  $V$  with respect to  $\varphi$  is any binary relation  $\preceq$  over  $V$  that satisfies  $\preceq \subseteq F_d(\preceq)$ . Similarly, a *reverse simulation* over  $V$  with respect to  $\varphi$  is any binary relation  $\preceq$  over  $V$  that satisfies  $\preceq \subseteq F_r(\preceq)$ .

In contrast to reverse simulations, direct simulations depend on a state formula  $\varphi$ . Also, the definition of direct simulation is inspired by the notion of *stable* relations above. Direct (reverse) simulations are usually denoted by  $\preceq_d$  ( $\preceq_r$ ). The following direct and reverse simulations are used as running examples throughout the paper.

*Example 1.* The empty relation  $a \preceq_\emptyset b := \text{false}$  is a direct and a reverse simulation.

*Example 2.* Equality ( $=$ ) between states is a direct and a reverse simulation.

*Example 3.* The relation  $s_1 \preceq_I s_2 := I(s_1) \wedge I(s_2)$  is a reverse simulation, where  $I$  is the predicate for describing the set of initial states of the given program.

*Example 4.* Now, consider programs  $\langle V, I, T \rangle$  with inputs such that  $\text{input}(x)$  holds iff  $x$  is an input variable. The relation

$$s_1 =_i s_2 := \text{for all variables } x \in V . \text{input}(x) \text{ or } s_1(x) = s_2(x),$$

with  $s(x)$  denoting the value of the variable  $x$  in the state  $s$ , is a reverse simulation, since the values of the input variables are not constrained by the predicate  $I$  and their next values are not constrained by  $T$ . Obviously, for transition systems with inputs, the relation  $s_1 =_i s_2$  is weaker than  $=$ , and therefore gives rise to shorter paths.

*Example 5.* We now consider timed automata programs and clock constraints. The region equivalence  $\sim_{TA}$ , which give rise to finitely many clock regions, is stable, and therefore a direct simulation.

The notions of direct and reverse simulation are modular in the sense that the union of direct (reverse) simulations is also a direct (reverse) simulation.

**Proposition 1 (Modularity).** If  $\preceq_1$  and  $\preceq_2$  are direct (reverse) simulations, then  $\preceq_1 \cup \preceq_2$  is also a direct (reverse) simulation.

This property follows directly from the definitions of direct (reverse) simulations in Definition 1 and from the monotonicity of the functors  $F_d$  and  $F_r$ . For example, the reverse simulations  $\preceq_I$  and  $=_i$  in Examples 3 and 4 may be combined to obtain a new reverse simulation.

Given a program  $M = \langle V, I, T \rangle$  and a property  $\varphi$ , the associated *largest direct (reverse) simulation* relation  $\preceq_D$  ( $\preceq_R$ ) is obtained as the greatest fixpoint of the functor  $F_d$  ( $F_r$ ) in Definition 1. These fixpoints exist, since  $F_d$  and  $F_r$  are monotonic. However, the fixpoint iterations are often prohibitively expensive, and a direct (reverse) simulation is only obtained on convergence of the iteration. The iteration in Proposition 2 provides a viable alternative in that a reverse (direct) simulation is refined to obtain a stronger reverse (direct) simulation. The proof of the proposition below follows from the definitions of reverse (direct) simulations, from the monotonicity of the functors  $F_r$  ( $F_d$ ), and from modularity (Proposition 1).

**Proposition 2 (Anytime Iteration).** If  $\preceq_r$  ( $\preceq_d$ ) is a reverse (direct) simulation, then for all  $n \geq 0$  the relation  $\preceq_{r,n}$  ( $\preceq_{d,n}$ ) is also a reverse (direct) simulation:

$$\begin{aligned} \preceq_{r,0} &:= \preceq_r & \preceq_{d,0} &:= \preceq_d \\ \preceq_{r,n} &:= \preceq_{r,n-1} \cup F_r(\preceq_{r,n-1}) & \preceq_{d,n} &:= \preceq_{d,n-1} \cup F_d(\preceq_{d,n-1}) \end{aligned}$$

Consequently, this iteration gives rise to an *anytime* algorithm for computing direct (reverse) simulations, and equality  $=$ , for example, may be used as seed, since it is both a direct and a reverse simulation (see Example 2). Also, quantifier elimination algorithms such as the one in Section 6 may be used in this iteration.

## 4 Completeness of $k$ -Induction

Given the notions of direct and reverse simulations, we develop sufficient conditions for proving completeness of  $k$ -induction. These results are based on restricting paths to not contain states that are similar with respect to a given *direct* or *reverse* simulation. For direct (reverse) simulations we define a compressed



**Fig. 1.** Incompleteness of  $k$ -induction.

path w.r.t. to the given direct (reverse) simulation as a path  $\pi(s_0, s_1, \dots, s_n)$  not containing any  $s_i, s_j$  with  $j < i$  ( $i < j$ ) such that  $s_i$  directly (reversely) simulates  $s_j$ .

**Definition 2 (Path Compression).**

- A path  $\pi \preceq^d(s_0, s_1, \dots, s_n)$  is *compressed* w.r.t. the direct simulation  $\preceq_d$  if:

$$\pi \preceq^d(s_0, s_1, \dots, s_n) := \pi(s_0, s_1, \dots, s_n) \wedge \bigwedge_{0 \leq j < i \leq n} s_i \not\preceq_d s_j.$$

- A path  $\pi \preceq^r(s_0, s_1, \dots, s_n)$  is *compressed* w.r.t. the reverse simulation  $\preceq_r$  if:

$$\pi \preceq^r(s_0, s_1, \dots, s_n) := \pi(s_0, s_1, \dots, s_n) \wedge \bigwedge_{0 \leq i < j \leq n} s_i \not\preceq_r s_j.$$

A path that is compressed with respect to the reverse and the direct simulations  $\preceq_r$  and  $\preceq_d$  is denoted by  $\pi \preceq^{r,d}$ .

For example, a path  $\pi(s_0, \dots, s_n)$  is compressed w.r.t. the reverse simulation ( $=$ ) from Example 2 iff it is acyclic. Moreover, given the reverse simulation  $\preceq_I$  from Example 3, a path  $\pi(s_0, \dots, s_n)$  is compressed w.r.t.  $\preceq_I$  iff it contains at most one initial state. Obviously, for transition systems with inputs, the relation ( $=_i$ ) (see Example 4) is weaker than ( $=$ ), and therefore give rise to shorter compressed paths. We have collected all ingredients for defining  $k$ -induction for arbitrarily compressed paths.

**Definition 3 ( $k$ -Induction).** Let  $M = \langle V, I, T \rangle$  be a program,  $k$  an integer,  $\preceq_r$  a reverse simulation, and  $\preceq_d$  a direct simulation. The induction scheme of depth  $k$ ,  $\text{IND}^{\preceq^{r,d}}(k)$  allows one to deduce the invariance of  $\varphi$  in  $M$  if the following holds.

- $I(s_0) \wedge \pi \preceq^{r,d}(s_0, \dots, s_{k-1}) \rightarrow \varphi(s_0) \wedge \dots \wedge \varphi(s_{k-1})$
- $\varphi(s_n) \wedge \dots \wedge \varphi(s_{n+k-1}) \wedge \pi \preceq^{r,d}(s_n, \dots, s_{n+k}) \rightarrow \varphi(s_{n+k})$

For example, given the empty relationship  $\preceq_\emptyset$  from Example 1,  $\text{IND}^{\preceq_\emptyset}$  reduces to the naive, incomplete  $k$ -induction on arbitrary paths. Consider, for example, the system in Figure 1 and a property  $\varphi$ , which is assumed to hold only in  $q_4$ . Now, the execution sequence  $\underbrace{q_3 \rightsquigarrow q_3 \rightsquigarrow \dots \rightsquigarrow q_3}_k \rightsquigarrow q_4$  is not  $k$ -inductive,

but it is ruled out under the acyclic path restriction. The complete  $k$ -induction schemes in [17], which consider only acyclic paths and paths that only visit ini-

tial states once can be recovered by instantiating Definition 3 with the relations  $(=)$  (Example 2) and  $(\preceq_I)$  (Example 3), respectively. Since both  $(=)$  and  $(\preceq_I)$  are reverse simulations, an induction scheme restricted to acyclic paths visiting initial states at most once is obtained by modularity (Proposition 1).

Completeness of  $k$ -induction relies heavily on the notion of path compression. We now state the main lemma.

**Lemma 1 (Compressing non- $\pi^{\preceq_{r,d}}$  paths).** Let  $\pi(s_0, \dots, s_n)$  be a given path; then:

1. There exists a  $\pi^{\preceq_r}$ - compressed path  $\pi^{\preceq_r}(q_0, \dots, q_m)$ , s.t.  $q_m = s_n$  and  $m \leq n$ .
2. There exists a  $\pi^{\preceq_d}$ - compressed path  $\pi^{\preceq_d}(q_0, \dots, q_m)$ , s.t.  $q_0 = s_0$  and  $m \leq n$ .

**Proofsketch.** Assume a path  $\pi(s_0, \dots, s_n)$ , which is not compressed w.r.t.  $\preceq_r$ . By Definition 1 it follows that there are states  $s_i, s_j \in \pi(s_0, \dots, s_n)$  such that  $s_i \preceq_r s_j$ , and  $i < j$ . We distinguish two cases. First, if  $s_i$  is an initial state, then so is  $s_j$ , and therefore a shorter path  $\pi(s_j, \dots, s_n)$  is obtained as a counterexample. Second, if  $s_i$  is not an initial state, then  $s_i \neq s_0$ , and there exists a  $s_{i-1}$  such that  $T(s_{i-1}, s_i)$ . Since  $s_i \preceq_r s_j$  it follows by Definition 1 that there is a state  $s'_{i-1}$ , such that  $s_{i-1} \preceq_r s'_{i-1}$  and  $T(s'_{i-1}, s_j)$ . If  $s_{i-1}$  is initial state, then so is  $s'_{i-1}$ , and since  $i < j$  a shorter path  $\pi^{\preceq_r}(s'_{i-1}, s_j, \dots, s_n)$  is obtained. If  $s_{i-1}$  is not initial, by repeating the above argument a shorter path is constructed. In both cases a shorter path is obtained, if such path is not a compressed path, then it is further reduced. The proof for  $\pi^{\preceq_d}$ - compressed paths works analogously.

$\text{IND}^{\preceq_{r,d}}(k)$  is *complete* if:  $\varphi$  is an invariant of  $M$  iff there is a  $k$  such that  $\text{IND}^{\preceq_{r,d}}(k)(\varphi)$ . Now, completeness of  $k$ -induction follows from the main lemma 1 above.

**Theorem 1 (Completeness).**  $\text{IND}^{\preceq_{r,d}}(k)$  is a complete proof method iff there is an upper bound on the length of the paths  $\pi^{\preceq_{r,d}}(s_0, \dots, s_n)$ .

Using the simulation from Example 2, Theorem 1 is instantiated to obtain the following complete  $k$ -induction for finite-state systems.

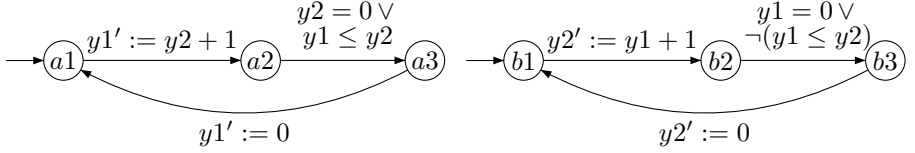
**Corollary 1.** Let  $M$  be a finite-state program over  $V$  and  $\varphi$  a state property in  $V$ ; then  $\text{IND}^=(k)$  induction is complete.

In general,  $k$ -induction for  $(=)$  is not complete for infinite-state systems. Consider, for example, the program  $M = \langle I, T \rangle$  over the integer state variable  $x$  with  $I = (x = 0)$  and  $T = (x' = x + 2)$ , and the formula  $x \neq 3$ . Obviously, it is the case that  $x \neq 3$  is invariant in  $M$ , but there exists no  $k \in \mathbb{N}$  such that the property is proven by  $\text{IND}^=(k)$ . However,  $k$ -induction is complete for timed automata, since the equivalence relation  $\sim_{TA}$  is a direct simulation (Example 5), and an upper bound on the length of the paths  $\pi^{\sim_{TA}}(s_0, \dots, s_n)$  is given by the number of clock regions.

**Corollary 2.** Let  $M$  be a timed automata program over the clock evaluations  $C$  and  $\varphi$  a clock constraint in  $C$ ; then  $\text{IND}^{\sim_{TA}}(k)$  induction is complete.

Similar results are obtained for other direct and reverse simulations and combinations thereof.





**Fig. 2.** Bakery Mutual Exclusion Protocol.

## 5 Invariant Strengthening

Whenever  $k$ -induction fails to prove a property  $\varphi$ , there is a counterexample  $\pi = s_n, s_{n+1}, \dots, s_{n+k}$  such that the first  $k$  states satisfy  $\varphi$  whereas the last state  $s_{n+k}$  does not satisfy this property. If  $s_n$  is indeed reachable, then  $\varphi$  is not invariant. Otherwise, the counterexample is labeled as *spurious* and it is inconclusive whether  $\varphi$  is invariant or not. However, by assuming  $s_n$  to be unreachable, such a spurious counterexample is used to obtain a strengthened invariant  $\varphi \wedge \neg(s_n)$ .

Consider, for example, the property  $\neg(q_4)$  for the system in Figure 1. Induction of depth  $k = 1$  fails, and the counterexample  $q_3 \rightsquigarrow q_4$  is obtained. Now,  $\neg(q_4)$  is strengthened to obtain  $\neg(q_4) \wedge \neg(q_3)$ , which is proven using 1-induction. More generally, whenever the induction step of  $\text{IND}^{\preceq_{r,d}}(k)$  fails, the formula  $Q(s_n, \dots, s_{n+k}) := \varphi(s_n) \wedge \dots \wedge \varphi(s_{n+k-1}) \wedge \pi^{\preceq_{r,d}}(s_n, \dots, s_{n+k}) \wedge \neg\varphi(s_{n+k})$  is satisfiable, and each satisfying assignment describes a counterexample for the induction step. Thus, we define the predicate  $U(s)$  for representing the set of possibly unreachable states, which may reach the bad state in  $k$  steps by means of a  $\pi^{\preceq_{r,d}}$  path,  $U(s) = \exists s_{n+1}, \dots, s_{n+k}. Q(s, s_{n+1}, \dots, s_{n+k})$ . Now,  $\varphi$  is strengthened as  $\varphi \wedge \neg U(s)$ , and quantifier elimination is used for transforming this strengthened formula into an equivalent Boolean constraint formula. For the general case, we use the quantifier elimination procedure in Section 6. Notice, however, that for special cases such as guarded command languages, the quantifiers in  $U(s)$  are eliminated using purely *syntactic* operations such as substitution, since all quantifications are over “next-state” variables  $x$  for which there are explicit solutions  $f(\cdot)$ . An example might help to illustrate the combination of  $k$ -induction, strengthening, and quantifier elimination.

*Example 6.* Consider the usual stripped-down version of Lamport’s Bakery protocol in Figure 2 with the initial value 0 for both counters  $y1$  and  $y2$  and the mutual exclusion property  $MX$  defined by  $\neg(pc1 = a3 \wedge pc2 = b3)$ . We apply 3-induction with the empty simulation relation  $\preceq_\emptyset$ . The base step holds and the induction step fails, thus we obtain

$$U(s_n) := \exists s_{n+1}, s_{n+2}, s_{n+3}. MX(s_n) \wedge MX(s_{n+1}) \wedge \\ MX(s_{n+2}) \wedge \pi^{\preceq_\emptyset}(s_n, s_{n+1}, s_{n+2}, s_{n+3}) \wedge \neg MX(s_{n+3})$$

with states  $s_i$  of the form  $(pc1_i, y1_i, pc2_i, y2_i)$ . Since the transitions of the Bakery protocol are in terms of guarded commands, simple substitution is used to obtain

a quantifier-eliminated form,  $R(s)$ , defined as

$$R(s) := (pc1 = a1 \wedge pc2 = b2 \wedge y2 = 0) \vee (pc1 = a2 \wedge pc2 = b1 \wedge y1 = 0).$$

Now, the strengthened property  $MX(s) \wedge \neg R(s)$  is proven using 3-induction.

## 6 Quantifier Elimination

Given a quantified formula  $\exists vars. \varphi$  with  $\varphi \in \text{Bool}(\mathcal{C})$ , quantifier-elimination procedures usually work by transforming  $\varphi$  into disjunctive normal form (DNF) and distributing the existential quantifiers over disjunctions. Thus, one is left with eliminating quantifiers from a set of existentially quantified conjunctions of literals. We assume as given such a procedure  $\mathcal{C}\text{-}qe$ . The main drawback of these procedures is that there is a potential exponential blowup in the initial transformation to DNF and  $\mathcal{C}\text{-}qe$  might even return further disjunctions (as is the case for Presburger arithmetic); this problem has been addressed for the Boolean case by McMillan [14].

The quantifier elimination problem for invariant strengthening, as discussed in Section 5 allows for a purely syntactic quantifier elimination as long as we are restricting ourselves to guarded command programs. In these cases,  $\mathcal{C}\text{-}qe$  just applies the *substitution rule* ( $x \notin vars(\psi)$ )

$$(\exists x. (x = \psi) \wedge \varphi(x)) \text{ iff } \varphi(\psi);$$

possibly followed by simplification. Quantifier elimination by substitution has already been used in the context of model checking, for example, by Coudert, Berthet, and Madre [15] and more recently by Williams, Biere, Clarke, Gupta [20], and Abdulla, Bjessé, Eén [1]. Another  $\mathcal{C}\text{-}qe$  function is used in McMillan's [14] quantifier elimination algorithm based on propositional SAT solving, in that his  $\mathcal{C}\text{-}qe(vars, c)$  simply deletes the literals in  $c$ , which contain a variable in  $vars$ . In contrast, depending on the background theory, arbitrary complex quantifier elimination procedures, such as the ones for Presburger arithmetic or real-closed fields, can also be used here.

As motivated above, the initial DNF computation should usually be avoided when possible. Given a set of existentially quantified variables  $vars$  and a quantifier-free formula  $\varphi$  in  $\text{Bool}(\mathcal{C})$ , the algorithm  $qe(vars, \varphi)$  in Figure 3 returns a formula in  $\text{Bool}(\mathcal{C})$  which is equivalent to  $\exists vars. \varphi$ . The procedure  $qe$  relies on a satisfiability solver for formulas  $\varphi \in \text{Bool}(\mathcal{C})$ , which is assumed to enumerate representations of sets of satisfiable models in terms of conjunctions of literals in  $\varphi$ . Such a solver is described, for example, in [7,6]. These solutions are supposed to be enumerated by successive calls to *next-solution* in Figure 3. Since there are only a finite number of solutions in terms of subsets of literals, the function  $qe$  is terminating. Moreover, minimal solutions or good over-approximations thereof, as produced by the lazy theorem proving algorithm [7,6], accelerate convergence.

The variable  $c$  in Figure 3 stores the current solution obtained by *next-solution*, and the procedure  $\mathcal{C}\text{-}qe$  applies quantifier elimination for conjunction. In

```

procedure  $qe(vars, \varphi)$ 
   $\psi := false$ 
  loop
     $c := next\_solution(\varphi)$ 
    if  $c = false$  then return  $\psi$ 
     $c' := \mathcal{C}\text{-}qe(vars, c)$ 
     $\psi := \psi \vee c'$ 
     $\varphi := \varphi \wedge \neg c'$ 

```

**Fig. 3.** Lazy Quantifier Elimination.

many cases,  $\mathcal{C}\text{-}qe$  just applies the *substitution rule* to remove quantified variables. In order to obtain the next set of solutions, we rule out the current solutions by updating  $\varphi$  with the value  $\neg c'$  instead of  $\neg c$ , since  $\neg c'$  is more restrictive.

Thus, the quantifier elimination procedure in Figure 3 avoids eager computation of a disjunctive normal form. Moreover, a solver for  $\text{Bool}(\mathcal{C})$  is used to guide the search for relevant “conjunctions” in  $\varphi$ . In this way, the  $qe$  algorithm extends the lazy theorem proving paradigm described in [7,6] to the case of first-order reasoning.

*Example 7.* Consider

$$\begin{aligned} & \exists x_1, y_1 ((x_0 = 1 \vee x_0 = 3 \vee y_0 > 1) \wedge x_1 = x_0 - 1 \wedge y_1 = y_0 + 1) \\ & \vee ((x_0 = -1 \vee x_0 = -3) \wedge x_1 = x_0 + 2 \wedge y_1 = y_0 - 1) \wedge x_1 < 0 \end{aligned}$$

A first satisfiable conjunction of literals is obtained by, say

$$c := y_0 > 1 \wedge x_1 = x_0 - 1 \wedge y_1 = y_0 + 1 \wedge x_1 < 0.$$

Now, application of the substitution rule yields  $c' := y_0 > 1 \wedge x_0 - 1 < 0$ , and, after updating  $\varphi$  with  $\neg c'$  a second solution is obtained as

$$c := x_0 = -3 \wedge x_1 = x_0 + 2 \wedge y_1 = y_0 - 1 \wedge x_1 < 0.$$

Again, applying the substitution rule, one gets  $c' := x_0 = -3 \wedge x_0 + 2 < 0$ , and, since there are no further solutions, the quantifier-eliminated formula is  $(y_0 > 1 \wedge x_0 - 1 < 0) \vee (x_0 = -3 \wedge x_0 + 2 < 0)$ .

## 7 Experiments

We describe some of our experiments with  $k$ -induction and invariant strengthening. Our benchmark examples include infinite-state systems such as communication protocols, timed automata and linear hybrid systems.<sup>1</sup> In particular, Table 1 contains experimental results for the Bakery protocol as described earlier, Simpson’s protocol [18] to avoid interference between concurrent reads and

<sup>1</sup> These benchmarks are available at <http://www.csl.sri.com/~demoura/cav03examples>

writes in a fully asynchronous system, well-known timed automata benchmarks such as the train gate controller and Fischer’s mutual exclusion protocol, and three linear hybrid automata benchmarks for water level monitoring, the leaking gas burner, and the multi-rate Fischer protocol. Timed automata and linear hybrid systems are encoded as in [19]. Starting with  $k = 1$  we increase  $k$  until  $k$ -induction succeeds. We are using invariant strengthening only in cases where *syntactic* quantifier elimination based on substitution suffices. In particular, we do not use strengthening for the timed and hybrid automata examples, that is, *C-qe* tries to apply the substitution rule, if the resulting satisfiability problems for Boolean combinations of linear arithmetic constraints are solved using the lazy theorem proving algorithm described in [7] and implemented in the ICS decision procedures [9].

System Name	Proved with $k$	Time	Refinements
Bakery Protocol	3	0.21	1
Simpson Protocol	2	0.16	2
Train Gate Controller	5	0.52	0
Fischer Protocol	4	0.71	0
Water Level Monitor	1	0.08	0
Leaking Gas Burner	6	1.13	0
Multi Rate Fischer	4	0.84	0

**Table 1.** Results for  $k$ -induction. Timings are in seconds.

The experimental results in Table 1 are obtained on a 2GHz Pentium-IV with 1Gb of memory. The second column in Table 1 lists the minimal  $k$  for which  $k$ -induction succeeds, the third column includes the total time (in seconds) needed for all inductions from 0 to  $k$ , and the fourth column the number of strengthenings. Timings do not include the one for quantifier elimination, since we restricted ourselves to syntactic quantifier elimination only. Notice that invariant strengthening is essential for the proofs of the Bakery protocol and Simpson’s protocol, since  $k$ -induction alone does not succeed for any  $k$ .

Simpson’s protocol for avoiding interference between concurrent reads and writes in a fully asynchronous system has also been studied using traditional model checking techniques. Using an explicit-state model checker, Rushby [16] demonstrates correctness of a finitary version of this potentially infinite-state problem. Whereas it took around 100 seconds for the model checker to verify this stripped-down problem,  $k$ -induction together with invariant strengthening proves the general problem in a fraction of a second. Moreover, other nontrivial problems such as correctness of Illinois and Futurebus cache coherence protocols, as given by [8], are easily established using 1-induction with only one round of strengthening.

## 8 Related Work

We restrict this comparison to work we think is most closely related to ours. Sheeran, Singh, and Stålmarck's [17] also use *k-induction*, but their approach is restricted to finite-state systems only. They consider *k-induction* restricted to acyclic paths and each path is constrained to contain at most one initial state. These inductions are simple instances of our general induction scheme based on reverse and direct simulations. Moreover, invariant strengthening is used here to decrease the minimal *k* for which *k-induction* succeeds.

Our path compression techniques can also be used to compute tight completeness thresholds for BMC. For example, a *compressed recurrence diameter* is defined as the smallest *n* such that  $I(s_0) \wedge \pi^{\preceq_{r,d}}(s_0, \dots, s_n)$  is unsatisfiable. Using equality (=) for the simulation relation, this formula is equivalent to the *recurrence diameter* in [4]. A tighter bound of the recurrence diameter, where values of input variables are ignored, is obtained by using the reverse simulation  $=_i$ . In this way, the results in [12] are obtained as specific instances in our general framework based on reverse and direct simulations. In addition, the *compressed diameter* is defined as the smallest *n* such that

$$I(s_0) \wedge \pi^{\preceq_{r,d}}(s_0, \dots, s_n) \wedge \bigwedge_{i=0}^{n-1} \neg \pi_i^{\preceq_{r,d}}(s_0, s_i)$$

is unsatisfiable, where  $\pi_i^{\preceq_{r,d}}(s_0, s_i) := \exists s_1, \dots, s_{i-1}. \pi^{\preceq_{r,d}}(s_0, s_1, \dots, s_{i-1}, s_i)$  holds if there is a relevant path from  $s_0$  to  $s_i$  with *i* steps. Depending on the simulation relation, this compressed diameter yields tighter bounds for the completeness thresholds than the ones usually used in BMC [4].

## 9 Conclusion

We developed a general *k-induction* scheme based on the notion of reverse and direct simulation, and we studied completeness of these inductions. Although any *k-induction* proof can be reduced to a 1-induction proof with invariant strengthening, there are certain advantages of using *k-induction*. In particular, bugs of length *k* are detected in the initial step, and the number of strengthenings required to complete a proof is reduced significantly. For example, a 1-induction proof of the Bakery protocol requires three successive strengthenings each of which produces 4 new clauses. There is, however, a clear trade-off between the additional cost of using *k-induction* and the number of strengthenings required in 1-induction, which needs to be studied further.

## References

1. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *LNCS*, pages 411–425. Springer-Verlag, 2000.

2. R. Alur. Timed automata. In *Computer-Aided Verification, CAV 1999*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22, 1999.
3. S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.
4. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zh. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579, 1999.
5. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
6. L. de Moura and H. Rueß. Lemmas on demand for satisfiability solvers. *Annals of Mathematics and Artificial Intelligence*, 2002. Accepted for publication.
7. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 438–455. Springer-Verlag, July 27–30 2002.
8. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification (CAV'00)*, pages 53–68, 2000.
9. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving. In *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.
10. S. M. German and B. Wegbreit. A synthesizer of inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, Mar. 1975.
11. S. M. Katz and Z. Manna. A heuristic approach to program verification. In N. J. Nilsson, editor, *Proceedings of the 3rd IJCAI*, pages 500–512, Stanford, CA, Aug. 1973. William Kaufmann.
12. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *Proceedings of VMCAI'03*, Jan. 2003.
13. 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):11–44, Jan. 1995.
14. K. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Computer-Aided Verification, CAV 2002*, volume 2404 of *LNCS*. Springer-Verlag, 2002.
15. O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.
16. J. Rushby. Model checking Simpson's four-slot fully asynchronous communication mechanism. Technical report, CSL, SRI International, Menlo Park, Menlo Park, CA, July 2002.
17. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. *LNCS*, 1954:108, 2000.
18. H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings, Part E: Computers and Digital Techniques*, 137(1):17–30, Jan. 1990.
19. M. Sorea. Bounded model checking for timed automata. In *Proceedings of MTCS 2002*, volume 68 of *Electronic Notes in Theoretical Computer Science*, 2002.
20. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*. Springer-Verlag, 2000.

# Reasoning with Temporal Logic on Truncated Paths

Cindy Eisner<sup>1</sup>, Dana Fisman<sup>1,2\*</sup>, John Havlicek<sup>3</sup>, Yoad Lustig<sup>1</sup>, Anthony McIsaac<sup>4</sup>, and David Van Campenhout<sup>5\*\*</sup>

<sup>1</sup> IBM Haifa Research Lab

<sup>2</sup> Weizmann Institute of Science

<sup>3</sup> Motorola, Inc.

<sup>4</sup> STMicroelectronics, Ltd.

<sup>5</sup> Verisity Design, Inc.

**Abstract.** We consider the problem of reasoning with linear temporal logic on *truncated* paths. A truncated path is a path that is finite, but not necessarily maximal. Truncated paths arise naturally in several areas, among which are incomplete verification methods (such as simulation or bounded model checking) and hardware resets. We present a formalism for reasoning about truncated paths, and analyze its characteristics.

## 1 Introduction

Traditional LTL semantics over finite paths [15] are defined for maximal paths in the model. That is, if we evaluate a formula over a finite path under traditional LTL finite semantics, it is because the last state of the path has no successor in the model. One of the consequences of extending LTL [16] to finite paths is that the *next* operator has to be split into a *strong* and a *weak* version [15]. The strong version, which we denote by  $X!\varphi$ , does not hold at the last state of a finite path, while the weak version, which we denote by  $X\varphi$ , does.

In this paper, we consider not only finite maximal paths, but finite *truncated* paths. A truncated path is a finite path that is not necessarily maximal. Truncated paths arise naturally in incomplete verification methods such as simulation or bounded model checking. There is also a connection to the problem of describing the behavior of hardware resets in temporal logic, since intuitively we tend to think of a reset as somehow cutting the path into two disjoint parts - a finite, truncated part up until the reset, and a possibly infinite, maximal part after the reset.

Methods of reasoning about finite maximal paths are insufficient for reasoning about truncated paths. When considering a truncated path, the user might want to reason about properties of the truncation as well as properties of the model. For instance, the user might want to specify that a simulation test goes on long enough to discharge all outstanding obligations, or, on the other hand, that an

---

\* The work of this author was supported in part by the John Von Neumann Minerva Center for the Verification of Reactive Systems.

\*\* E-mail addresses: eisner@il.ibm.com (C. Eisner), dana@wisdom.weizmann.ac.il (D. Fisman), john.havlicek@motorola.com (J. Havlicek), yoadl@il.ibm.com (Y. Lustig), anthony.mcisaac@st.com (A. McIsaac), dvc@verisity.com (D. Van Campenhout).

obligation need not be met if it “is the fault of the test” (that is, if the test is too short). The former approach is useful for a test designed (either manually or by other means) to continue until correct output can be confirmed. The latter approach is useful for a test which has no “opinion” on the correct length of a test - for instance, a monitor running concurrently with the main test to check for bus protocol errors.

At first glance, it seems that the strong operators ( $X!$  and  $U$ ) can be used in the case that all outstanding obligations must be met, and the weak operators ( $X$  and  $W$ ) in the case that they need not. However, we would like a specification to be independent of the verification method used. Thus, for instance, for a specification  $[p \ U \ q]$ , we do not want the user to have to modify the formula to  $[p \ W \ q]$  just because she is running a simulation.

In such a situation, we need to define the semantics over a truncated path. In other words, at the end of the truncated path, the truth value must be decided. If the path was truncated after the evaluation of the formula completed, the truth value is already determined. The problem is to decide the truth value if the path was truncated before the evaluation of the formula completed, i.e., where there is *doubt* regarding what would have been the truth value if the path had not been truncated. For instance, consider the formula  $Fp$  on a truncated path such that  $p$  does not hold for any state. Another example is the formula  $Gq$  on a truncated path such that  $q$  holds for every state. In both cases we cannot be sure whether or not the formula holds on the original untruncated path.

We term a decision to return *true* when there is doubt the *weak view* and a decision to return *false* when there is doubt the *strong view*. Thus in the weak view the formula  $Fp$  holds for any finite path, while  $Gq$  holds only if  $q$  holds at every state on the path. And in the strong view the formula  $Fp$  holds only if  $p$  holds at some state on the path, while the formula  $Gq$  does not hold for any finite path. Alternatively, one can take the position that one should demand the maximum that can be reasonably expected from a finite path. For formulas of the form  $Fp$ , a prefix on which  $p$  holds for some state on the path is sufficient to show that the formula holds on the entire path, thus it is reasonable to demand that such a prefix exist. In the case of formulas of the form  $Gq$ , no finite prefix can serve as evidence that the formula holds on the entire path, thus requiring such evidence is not reasonable. Under this approach, then, the formula  $Fp$  holds only if  $p$  holds at some state on the path, while the formula  $Gq$  holds only if  $q$  holds at every state on the path. This is exactly the traditional LTL semantics over finite paths [15], which we term the *neutral view*.

In this paper, we present a semantics for LTL over truncated paths based on the weak, neutral, and strong views. We study properties of the *truncated semantics* for the resulting logic  $LTL^{trunc}$ , as well as its relation to the *informative prefixes* of [12]. We examine the relation between truncated paths and hardware resets, and show that our truncated semantics are mathematically equivalent to the *reset semantics* of [3].

The remainder of this paper is structured as follows. Section 2 presents our *truncated semantics*. Section 3 studies properties of our logic as well as its relation



to the *informative prefixes* of [12]. Section 4 shows the relation to hardware resets. Section 5 discusses related work. Section 6 concludes.

## 2 The Truncated Semantics

Recall that LTL is the logic with the following syntax:

**Definition 1** (LTL formulas).

- Every atomic proposition is an LTL formula.
- If  $\varphi$  and  $\psi$  are LTL formulas then the following are LTL formulas:
  - $\neg\varphi$       •  $\varphi \wedge \psi$       •  $X!\varphi$       •  $[\varphi \text{ U } \psi]$

Additional operators are defined as syntactic sugaring of the above operators:

- $\varphi \vee \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \wedge \neg\psi)$       •  $\varphi \rightarrow \psi \stackrel{\text{def}}{=} \neg\varphi \vee \psi$       •  $X\varphi \stackrel{\text{def}}{=} \neg(X!\neg\varphi)$
- $F\varphi \stackrel{\text{def}}{=} [\text{true} \text{ U } \varphi]$       •  $G\varphi \stackrel{\text{def}}{=} \neg F\neg\varphi$       •  $[\varphi \text{ W } \psi] \stackrel{\text{def}}{=} [\varphi \text{ U } \psi] \vee G\varphi$

According to our motivation presented above, the formula  $\varphi$  holds on a truncated path in the weak view if up to the point where the path ends, “nothing has yet gone wrong” with  $\varphi$ . It holds on a truncated path in the neutral view according to the standard LTL semantics for finite paths. In the strong view,  $\varphi$  holds on a truncated path if everything that needs to happen to convince us that  $\varphi$  holds on the original untruncated path has already occurred. Intuitively then, our truncated semantics are related to those of standard LTL on finite paths as follows: the weak view weakens all operators (e.g.  $\text{U}$  acts like  $\text{W}$ ,  $X!$  like  $X$ ), the neutral view leaves them unchanged, and the strong view strengthens them (e.g.  $\text{W}$  acts like  $\text{U}$ ,  $X$  like  $X!$ ).

We define the truncated semantics of LTL formulas over words<sup>1</sup> from the alphabet  $2^P$ . A letter is a subset of the set of atomic propositions  $P$  such that *true* belongs to the subset and *false* does not. We will denote a letter from  $2^P$  by  $\ell$  and an empty, finite, or infinite word from  $2^P$  by  $w$ . We denote the length of word  $w$  as  $|w|$ . An empty word  $w = \epsilon$  has length 0, a finite word  $w = (\ell_0\ell_1\ell_2\cdots\ell_n)$  has length  $n + 1$ , and an infinite word has length  $\infty$ . We denote the  $i^{\text{th}}$  letter of  $w$  by  $w^{i-1}$  (since counting of letters starts at zero). We denote by  $w^{i\cdots}$  the suffix of  $w$  starting at  $w^i$ . That is,  $w^{i\cdots} = (w^iw^{i+1}\cdots w^n)$  or  $w^{i\cdots} = (w^iw^{i+1}\cdots)$ . We denote by  $w^{i\cdots j}$  the finite sequence of letters starting from  $w^i$  and ending in  $w^j$ . That is,  $w^{i\cdots j} = (w^iw^{i+1}\cdots w^j)$ .

We make use of an “overflow” and “underflow” for the indices of  $w$ . That is,  $w^{j\cdots} = \epsilon$  if  $j \geq |w|$ , and  $w^{j\cdots k} = \epsilon$  if  $j \geq |w|$  or  $k < j$ . For example, in the semantics of  $[\varphi \text{ U } \psi]$  under weak context, when we say “ $\exists k$ ”,  $k$  is not required to be less than  $|w|$ .

The truncated semantics of an LTL formula are defined with respect to finite or infinite words and a context indicating the *strength*, which can be either

<sup>1</sup> Relating the semantics over words to semantics over models is done in the standard way. Due to lack of space, we omit the details.

weak, neutral or strong. Under the neutral context only non-empty words are evaluated; under weak/strong contexts, empty words are evaluated as well. We use  $w \models^S \varphi$  to denote that  $\varphi$  is satisfied under the model  $(w, S)$ , where  $S$  is “−” if the context is weak, null if it is neutral, and “+” if it is strong. We use  $w$  to denote an empty, finite, or infinite word,  $\varphi$  and  $\psi$  to denote LTL formulas,  $p$  to denote an atomic proposition, and  $j$  and  $k$  to denote natural numbers.

**holds weakly:** For  $w$  such that  $|w| \geq 0$ ,

1.  $w \models^- p \iff |w| = 0 \text{ or } p \in w^0$
2.  $w \models^- \neg \varphi \iff w \not\models^- \varphi$
3.  $w \models^- \varphi \wedge \psi \iff w \models^- \varphi \text{ and } w \models^- \psi$
4.  $w \models^- X! \varphi \iff w^{1..} \models^- \varphi$
5.  $w \models^- [\varphi U \psi] \iff \exists k \text{ such that } w^{k..} \models^- \psi, \text{ and for every } j < k, w^{j..} \models^- \varphi$

**holds neutrally:** For  $w$  such that  $|w| > 0$ ,

1.  $w \models \varphi \iff \varphi \in w^0$
2.  $w \models \neg \varphi \iff w \not\models \varphi$
3.  $w \models \varphi \wedge \psi \iff w \models \varphi \text{ and } w \models \psi$
4.  $w \models X! \varphi \iff |w| > 1 \text{ and } w^{1..} \models \varphi$
5.  $w \models [\varphi U \psi] \iff \exists k < |w| \text{ such that } w^{k..} \models \psi, \text{ and for every } j < k, w^{j..} \models \varphi$

**holds strongly:** For  $w$  such that  $|w| \geq 0$ ,

1.  $w \models^+ p \iff |w| > 0 \text{ and } p \in w^0$
2.  $w \models^+ \neg \varphi \iff w \not\models^+ \varphi$
3.  $w \models^+ \varphi \wedge \psi \iff w \models^+ \varphi \text{ and } w \models^+ \psi$
4.  $w \models^+ X! \varphi \iff w^{1..} \models^+ \varphi$
5.  $w \models^+ [\varphi U \psi] \iff \exists k \text{ such that } w^{k..} \models^+ \psi, \text{ and for every } j < k, w^{j..} \models^+ \varphi$

Our goal was to give a semantics to LTL formulas for truncated paths, but we have actually ended up with two parallel semantics: the neutral semantics, and the weak/strong semantics. The weak/strong semantics form a coupled dual pair because the negation operator switches between them. Before analyzing these semantics, we first unify them by augmenting LTL with truncate operators that connect the neutral semantics to the weak/strong semantics. Intuitively, `trunc_w` truncates a path using the weak view, while `trunc_s` truncates using the strong view. Formally,  $\text{LTL}^{\text{trunc}}$  is the following logic, where we use the term *boolean expression* to refer to any application of the standard boolean operators to atomic propositions, and we associate satisfaction of a boolean expression over a letter  $w^i$  with satisfaction of the boolean expression over the word  $w^{i..i}$ .

**Definition 2** ( $\text{LTL}^{\text{trunc}}$  formulas).

- Every atomic proposition is an  $\text{LTL}^{\text{trunc}}$  formula.

- If  $\varphi$  and  $\psi$  are  $\text{LTL}^{\text{trunc}}$  formulas and  $b$  is a boolean expression, then the following are  $\text{LTL}^{\text{trunc}}$  formulas:
  - $\neg\varphi$       •  $\varphi \wedge \psi$       •  $X!\varphi$       •  $[\varphi \text{ U } \psi]$       •  $\varphi \text{ trunc\_w } b$

We also add the dual of the `trunc_w` operator as syntactic sugar as follows:

$$\varphi \text{ trunc\_s } b \stackrel{\text{def}}{=} \neg(\neg\varphi \text{ trunc\_w } b)$$

The semantics of the standard LTL operators are as presented above. The semantics of the truncate operator are as follows:

- $w \models^- \varphi \text{ trunc\_w } b \iff w \models^- \varphi \text{ or } \exists k < |w| \text{ s.t. } w^k \models b \text{ and } w^{0..k-1} \models^- \varphi$
- $w \models \varphi \text{ trunc\_w } b \iff w \models \varphi \text{ or } \exists k < |w| \text{ s.t. } w^k \models b \text{ and } w^{0..k-1} \models^- \varphi$
- $w \models^+ \varphi \text{ trunc\_w } b \iff w \models^+ \varphi \text{ or } \exists k < |w| \text{ s.t. } w^k \models b \text{ and } w^{0..k-1} \models^- \varphi$

Thus, `trunc_w` performs a truncation and takes us to the weak view, and, as we show below, `trunc_s` performs a truncation and takes us to the strong view. There is no way to get from the weak/strong views back to the neutral view. This corresponds with our intuition that once a path has been truncated, there is no way to “untruncate” it.

### 3 Characteristics of the Truncated Semantics

In this section, we study properties of the truncated semantics as well as its relation to the *informative prefixes* of [12]. All theorems are given here without proof; the proofs can be found in the full version of the paper. We first examine relations between the views. The first theorem assures that the strong context is indeed stronger than the neutral, while the neutral is stronger than the weak.

**Theorem 3 (Strength relation theorem).** *Let  $w$  be a non-empty word.*

1.  $w \models^+ \varphi \implies w \models \varphi$
2.  $w \models \varphi \implies w \models^- \varphi$

The proof, obtained by induction on the structure of the formula, relies on the following lemma.

**Lemma 4** *Let  $\varphi$  be a formula in  $\text{LTL}^{\text{trunc}}$ . Then both  $\epsilon \models^- \varphi$  and  $\epsilon \not\models^+ \varphi$ .*

The following corollary to Theorem 3 states that for infinite paths, the weak/neutral/strong views are the same. Recall that the neutral view without the `trunc_w` operator is that of standard LTL over finite and infinite paths. Thus, for  $\text{LTL}^{\text{trunc}}$  formulas with no truncation operators (that is, for LTL formulas), Corollary 5 implies that all three views are equivalent over infinite paths to standard LTL semantics.

**Corollary 5** *If  $w$  is infinite, then  $w \models^- \varphi$  iff  $w \models \varphi$  iff  $w \models^+ \varphi$ .*

Intuitively, a truncated path  $w$  satisfies  $\varphi$  in the weak view if  $w$  “carries no evidence against”  $\varphi$ . It should then follow that any prefix of  $w$  “carries no evidence against”  $\varphi$ . Similarly,  $w$  satisfies  $\varphi$  in the strong view if it “supplies all the evidence needed” to conclude that  $\varphi$  holds on the original untruncated path. Hence any extension of  $w$  should also “supply all evidence needed” for this conclusion. The following theorem confirms these intuitive expectations. We first formalize the notions of prefix and extension.

**Definition 6 (Prefix, extension).**

$u$  is a prefix of  $v$ , denoted  $u \preceq v$ , if there exists a word  $u'$  such that  $uu' = v$ .  
 $w$  is an extension of  $v$ , denoted  $w \succeq v$ , if there exists a word  $v'$  such that  $vv' = w$ .

**Theorem 7 (Prefix/extension theorem).**

1.  $v \models^- \varphi \iff \forall u \preceq v, u \models^- \varphi$
2.  $v \models^+ \varphi \iff \forall w \succeq v, w \models^+ \varphi$

We now examine our intuitions regarding some derived operators. Since the `trunc_w` operator takes us to the weak view, we expect the `trunc_s` operator to take us to the strong view. The following observation confirms our intuition by capturing directly the semantics of the `trunc_s` operator.

**Observation 8**

- $w \models^- \varphi \text{ trunc}_s b \iff w \models^- \varphi \text{ and } \forall k < |w| \text{ if } w^k \models b \text{ then } w^{0..k-1} \models^+ \varphi$
- $w \models \varphi \text{ trunc}_s b \iff w \models \varphi \text{ and } \forall k < |w| \text{ if } w^k \models b \text{ then } w^{0..k-1} \models^+ \varphi$
- $w \models^+ \varphi \text{ trunc}_s b \iff w \models^+ \varphi \text{ and } \forall k < |w| \text{ if } w^k \models b \text{ then } w^{0..k-1} \models^+ \varphi$

The following observation shows that our intuitions regarding  $F$  and  $G$  on truncated paths hold. In particular,  $F\varphi$  holds for any formula  $\varphi$  in weak context on a truncated path, and  $G\varphi$  does not hold for any formula  $\varphi$  in strong context on a truncated path.

**Observation 9**

- $w \models^- F\varphi \iff \exists k \text{ s.t. } w^{k..} \models^- \varphi$
- $w \models^- G\varphi \iff \forall k, w^{k..} \models^- \varphi$
- $w \models F\varphi \iff \exists k < |w| \text{ s.t. } w^{k..} \models \varphi$
- $w \models G\varphi \iff \forall k < |w|, w^{k..} \models \varphi$
- $w \models^+ F\varphi \iff \exists k \text{ s.t. } w^{k..} \models^+ \varphi$
- $w \models^+ G\varphi \iff \forall k, w^{k..} \models^+ \varphi$

Note that for  $k \geq |w|$ ,  $w^{k..} = \epsilon$  and by Lemma 4,  $\epsilon \models^- \varphi$  and  $\epsilon \not\models^+ \varphi$  for every  $\varphi$ . Thus Observation 9 shows that for every formula  $\varphi$  and for every finite word  $w$ ,  $w \models^- F\varphi$  and  $w \not\models^+ G\varphi$ .

We have already seen that for infinite words, the semantics of the weak/neutral/strong contexts are equivalent and, in the absence of truncation operators, are the same as those of standard LTL. The following observations show that for finite words, the strength of an operator matters only in the neutral context since in a weak context every operator is weak ( $U$  acts like  $W$  and  $X!$  acts like  $X$ ) and in a strong context every operator is strong ( $W$  acts like  $U$  and  $X$  acts like  $X!$ ).

**Observation 10** *Let  $w$  be a finite word.*

- $w \models X\varphi \iff w \models \neg(X! \neg\varphi)$
- $w \models [\varphi U\psi] \iff w \models \neg[\neg\psi W(\neg\varphi \wedge \neg\psi)]$
- $w \models^+ X\varphi \iff w \models^+ X! \varphi$
- $w \models^+ [\varphi U\psi] \iff w \models^+ [\varphi W\psi]$
- $w \models^- X\varphi \iff w \models^- X! \varphi$
- $w \models^- [\varphi U\psi] \iff w \models^- [\varphi W\psi]$

A consequence of this is that under weak context it might be the case that both  $\varphi$  and  $\neg\varphi$  hold, while under strong context it might be the case that neither  $\varphi$  nor  $\neg\varphi$  holds. It follows immediately that  $\varphi \wedge \neg\varphi$  may hold in the weak context, while  $\varphi \vee \neg\varphi$  does not necessarily hold in the strong context. For example, let  $\varphi = \mathbf{XX}p$ . Then on a path  $w$  of length 1,  $w \models^- \varphi \wedge \neg\varphi$ , and  $w \not\models^+ \varphi \vee \neg\varphi$ . This property of the truncated semantics is reminiscent of a similar property in intuitionistic logic [6], in which  $\varphi \vee \neg\varphi$  does not necessarily hold.

We now argue that the truncated semantics formalizes the intuition behind the weak, neutral and strong views. Recall that one of the motivating intuitions for the truncated semantics is that if a path is truncated before evaluation of  $\varphi$  “completes”, then the truncated path satisfies  $\varphi$  weakly but does not satisfy  $\varphi$  strongly. If the evaluation of  $\varphi$  “completes” before the path is truncated, then the truth value on the truncated path is the result of the evaluation. Thus, in order to claim that we capture the intuition we need to define when the evaluation of a formula completes. In other words, given a word  $w$  and a formula  $\varphi$  we would like to detect the shortest prefix of  $w$  which suffices to conclude that  $\varphi$  holds or does not hold on  $w$ . We call such a prefix the *definitive prefix* of  $\varphi$  with respect to  $w$ .

**Definition 11 (Definitive prefix).** *Let  $w$  be a non-empty path and  $\varphi$  a formula. The definitive prefix of  $w$  with respect to  $\varphi$ , denoted  $dp(w, \varphi)$ , is the shortest finite prefix  $u \preceq w$  such that*

$$u \models^- \varphi \iff u \models \varphi \iff u \models^+ \varphi$$

*if such  $u$  exists and  $\top$  otherwise.*

Intuitively, if  $w$  is finite and  $dp(w, \varphi) = \top$ , then even after examination of all of  $w$ , our decision procedure leaves doubt about the dispositions of both  $\varphi$  and  $\neg\varphi$  on  $w$ . Therefore, both are satisfied weakly on  $w$ , neither is satisfied strongly on  $w$ , and all of  $w$  is needed to determine which one is satisfied neutrally on  $w$ . If  $dp(w, \varphi) \neq \top$ , then for finite or infinite  $w$ , examination of  $dp(w, \varphi)$  is exactly enough for our decision procedure to resolve without doubt the truth value of  $\varphi$  over any prefix  $v$  of  $w$  such that  $v \succeq dp(w, \varphi)$ . Therefore, any proper prefix of  $dp(w, \varphi)$  satisfies weakly both  $\varphi$  and  $\neg\varphi$ , while  $dp(w, \varphi)$  satisfies strongly exactly one of  $\varphi$  or  $\neg\varphi$ , as do all of its extensions. The following theorem states this formally:

**Theorem 12 (Definitive prefix theorem).** *Let  $v$  be a non-empty word and  $\varphi$  an LTL<sup>trunc</sup> formula.*

- If  $dp(v, \varphi) \neq \top$  then
  - $u \prec dp(v, \varphi) \implies u \models^- \varphi$  and  $u \models^- \neg\varphi$

- $u \succeq dp(v, \varphi) \implies u \models^+ \varphi$  or  $u \models^+ \neg\varphi$
- Otherwise
  - for every finite  $u \preceq v$ , ( $u \models^- \varphi$  and  $u \models^- \neg\varphi$ ) and ( $u \not\models^+ \varphi$  and  $u \not\models^+ \neg\varphi$ )

Plainly,  $dp(w, \varphi) = dp(w, \neg\varphi)$ . If  $u$  is the definitive prefix of  $w$  with respect to  $\varphi$ , then it is its own definitive prefix with respect to  $\varphi$ . That is:

**Proposition 13** *Let  $w$  be a non-empty word and  $\varphi$  an  $\text{LTL}^{\text{trunc}}$  formula. Then*

$$dp(w, \varphi) \neq \top \implies dp(w, \varphi) = dp(dp(w, \varphi), \varphi)$$

The definitive prefix of the truncated semantics is closely related to the concept of informative prefix in [12]. That work examines the problem of model checking safety formulas for standard LTL over maximal paths. Let a *safety formula* be a formula  $\varphi$  such that any path  $w$  violating  $\varphi$  contains a prefix  $w^{0..k}$  all of whose infinite extensions violate  $\varphi$  [15]. Such a prefix is termed a *bad prefix* by [12]. Our intuitive notion of a bad prefix says that it should be enough to fully explain the failure of a safety formula. However, [12] showed that for LTL over maximal paths, there are safety formulas for which this does not hold. For instance, consider the formula  $\varphi = (G(q \vee FGp) \wedge G(r \vee FG\neg p)) \vee Gq \vee Gr$ . In standard LTL semantics,  $\varphi$  is equivalent to  $Gq \vee Gr$ , and the bad prefixes are exactly the finite words satisfying  $\neg(Gq \vee Gr)$ . However, we somehow feel that such a prefix is too short to “tell the whole story” of formula  $\varphi$  on path  $w$ , because it does not explain that  $(FGp) \wedge (FG\neg p)$  is unsatisfiable.

The concept of a prefix which tells the whole story regarding the failure of formula  $\varphi$  on path  $w$  is formalized by [12] as an *informative prefix*. The precise definition in [12] is inductive over the finite path and the structure of  $\neg\varphi$ , which is assumed to be in positive normal form. The definition accomplishes an accounting of the discharge of the various sub-formulas of  $\neg\varphi$  and is omitted due to lack of space. From the intuitive description, if  $u$  is an informative prefix for  $\varphi$ , then we should have that  $u \models^+ \neg\varphi$ , or equivalently,  $u \not\models^+ \varphi$ . The following theorem confirms this expectation and its converse.

**Theorem 14 (Informative prefix theorem).** *Let  $w$  be a non-empty finite word and  $\varphi$  an LTL formula.*

$$w \not\models^+ \varphi \iff w \text{ is informative for } \varphi$$

Notice that Theorem 14 shows that the notion of informative prefix for  $\varphi$ , defined in terms of syntactic structure, is captured semantically by the weak/strong truncated semantics. Furthermore, the definitive prefix does not require formulas to be in positive normal form, as does the informative prefix, and is symmetric in  $\varphi$  and  $\neg\varphi$ , as opposed to the informative prefix, which is defined only for formulas that do not hold. The precise relation of definitive prefixes to informative prefixes is given by the following corollary.

**Corollary 15** *Let  $w$  be a non-empty path and let  $\varphi$  be an LTL formula.*

*If  $dp(w, \varphi) = \top$ , then  $w$  has no informative prefix for either  $\varphi$  or  $\neg\varphi$ .*

*Otherwise,  $dp(w, \varphi)$  is the shortest informative prefix of  $w$  for either  $\varphi$  or  $\neg\varphi$ .*

## 4 Relation to Hardware Resets

There is an intimate relation between the problem of hardware resets and that of truncated vs. maximal paths: a hardware reset can be viewed as truncating the path and canceling future obligations; thus it corresponds to the weak view of truncated paths. We now consider the relation between the semantics given to the hardware reset operators of ForSpec [3] (termed the *reset semantics* by [2]) and of Sugar2.0 [8] (termed the *abort semantics* by [2]) and the truncated semantics we have presented above. We show that the truncated semantics are equivalent to the reset semantics, thus by [2], different from the abort semantics.

**Reset Semantics** The reset semantics are defined as follows, where [3] uses `accept_on` as the name of the `trunc_w` operator. Let  $a$  and  $r$  be mutually exclusive boolean expressions, where  $a$  is the condition for truncating a path and accepting the formula, and  $r$  is the condition for rejection. Let  $w$  be a non-empty word<sup>2</sup>. As before, we use  $\varphi$  and  $\psi$  to denote  $LTL^{trunc}$  formulas,  $p$  to denote an atomic proposition, and  $j$  and  $k$  to denote natural numbers. The reset semantics are defined in terms of a four-way relation between words, contexts  $a$  and  $r$ , and formulas, denoted  $\models_{\mathcal{R}}$ . The definition of the reset semantics makes use of a two-way relation between letters and boolean expressions which is defined in the obvious manner.

1.  $\langle w, a, r \rangle \models_{\mathcal{R}} p \iff w^0 \models_{\mathcal{R}} a \vee (p \wedge \neg r)$
2.  $\langle w, a, r \rangle \models_{\mathcal{R}} \neg \varphi \iff \langle w, r, a \rangle \not\models_{\mathcal{R}} \varphi$
3.  $\langle w, a, r \rangle \models_{\mathcal{R}} \varphi \wedge \psi \iff \langle w, a, r \rangle \models_{\mathcal{R}} \varphi$  and  $\langle w, a, r \rangle \models_{\mathcal{R}} \psi$
4.  $\langle w, a, r \rangle \models_{\mathcal{R}} X! \varphi \iff w^0 \models_{\mathcal{R}} a$  or  $(w^0 \not\models_{\mathcal{R}} r$  and  $|w| > 1$  and  $\langle w^{1..}, a, r \rangle \models_{\mathcal{R}} \varphi)$
5.  $\langle w, a, r \rangle \models_{\mathcal{R}} [\varphi \cup \psi] \iff$  there exists  $k < |w|$  such that  $\langle w^{k..}, a, r \rangle \models_{\mathcal{R}} \psi$ , and for every  $j < k$ ,  $\langle w^{j..}, a, r \rangle \models_{\mathcal{R}} \varphi$
6.  $\langle w, a, r \rangle \models_{\mathcal{R}} \varphi \text{ trunc\_w } b \iff \langle w, a \vee (b \wedge \neg r), r \rangle \models_{\mathcal{R}} \varphi$

**Abort Semantics** The abort semantics are defined in [8] as the traditional LTL semantics over finite and infinite (non-empty) paths, with the addition of a truncate operator (termed there `abort`), as follows, where we use  $\models_{\mathcal{A}}$  to denote satisfaction under these semantics:

$$w \models_{\mathcal{A}} \varphi \text{ trunc\_w } b \iff \text{either } w \models_{\mathcal{A}} \varphi \text{ or there exist } j < |w| \text{ and word } w' \text{ such} \\ \text{that } w^j \models_{\mathcal{A}} b \text{ and } w^{0..j-1}w' \models_{\mathcal{A}} \varphi$$

Intuitively, the reset and abort semantics are very similar. They both specify that the path up to the point of reset must be “well behaved”, without regard to

<sup>2</sup> In [3], the reset semantics are defined over infinite words. We present a straightforward extension to (non-empty) finite as well as infinite words.

the future behavior of the path. The difference is in the way future obligations are treated, and is illustrated by the following formulas:

$$(G(p \rightarrow F(\varphi \wedge \neg\varphi))) \text{ trunc\_w } b \quad (1)$$

$$(G\neg p) \text{ trunc\_w } b \quad (2)$$

Formulas 1 and 2 are equivalent in the abort semantics, because the future obligation  $\varphi \wedge \neg\varphi$  is not satisfiable. They are not equivalent in the reset semantics, because the reset semantics “do not care” that  $\varphi \wedge \neg\varphi$  is not satisfiable. Thus there exist values of  $w$ ,  $a$ , and  $r$  such that Formula 1 holds under the reset semantics, while Formula 2 does not. For example, consider a word  $w$  such that  $p$  holds on  $w^5$  and for no other letter and  $b$  holds on  $w^6$  and on no other letter. If  $a = r = \text{false}$ , then Formula 1 holds on word  $w$  in the reset semantics under contexts  $a$  and  $r$ , while Formula 2 does not.

As shown in [2], the difference between the reset and the abort semantics causes a difference in complexity. While the complexity of model checking the reset semantics is PSPACE-complete, the complexity of model checking the abort semantics is SPACE( $\exp(k, n)$ )-complete where  $n$  is the length of the formula and  $k$  is the nesting depth of `trunc_w`.

Unlike the abort semantics, the truncated and reset semantics make no existential requirements of a path after truncation. The truncated semantics discard the remainder of the path after truncation, while the reset semantics accumulate the truncate conditions for later use. Theorem 16 states that they are the same.

**Theorem 16 (Equivalence theorem).** *Let  $\varphi$  be a formula of  $\text{LTL}^{\text{trunc}}$ ,  $a$  and  $r$  mutually exclusive boolean expressions, and  $w$  a non-empty word. Then,*

$$\langle w, a, r \rangle \models_{\mathcal{R}} \varphi \iff w \models (\varphi \text{ trunc\_w } a) \text{ trunc\_s } r$$

In particular, for an infinite  $w$ ,  $\langle w, \text{false}, \text{false} \rangle \models_{\mathcal{R}} \varphi$  in the reset semantics if and only if  $w \models \varphi$  in the truncated semantics. It follows easily that the truncated semantics are not more expressive or harder to decide than the reset semantics, which were shown in [2] to have the same expressiveness and complexity as LTL.

## 5 Related Work

Semantics for LTL is typically defined over infinite paths. Often finite paths are dealt with by infinitely repeating the last state (see e.g. [7]). Lichtenstein et al. [14] were the first to extend the semantics of LTL to finite paths. In particular, they introduced the *strong next* operator (see also [13],[15, pages 272-273]). However, they consider only finite maximal paths, and the issue of truncated paths is not considered.

The issue of using temporal logic specifications in simulation is addressed by [1]. They consider only a special class of safety formulas [4] which can be translated into formulas of the form  $Gp$ , and do not distinguish between maximal and truncated paths.



The idea that an obligation need not be met in the weak view if it “is the fault of the test” is directly related to the idea of weak clocks in [8], in which obligations need not be met if it “is the fault of the clock”. The weak/strong clocked semantics of [8] were the starting point for investigations that have led to [9], which proposes a clocked semantics in which the clock is strengthless, and to the current work, which preserves much of the intuition of the weak/strong clocked semantics in a simpler, unlocked setting.

The work described here is the result of discussions in the LRM sub-committee of the Accellera Formal Verification Technical Committee. Three of the languages (Sugar2.0, ForSpec, CBV [10]) examined by the committee enhance temporal logic with operators intended to support hardware resets. We have discussed the reset and abort semantics of ForSpec and Sugar2.0 in detail. The operator of CBV, while termed `abort`, has semantics similar to those of ForSpec’s `accept_on/reject_on` operators. As we have shown, our truncated semantics are mathematically equivalent to the reset semantics of ForSpec. However, the reset semantics take the operational view in that they tell us in a fairly direct manner how to construct an alternating automaton for a formula. Our approach takes the denotational view and thus tells us more directly the effect of truncation on the formula. This makes it easy to reason about the semantics in a way that is intuitively clear, because we can reason explicitly about three constant contexts (weak/neutral/strong) which are implicit in the operational view.

Bounded model checking [5] considers the problem of searching for counterexamples of finite length to a given LTL formula. The method is to solve the existential model checking problem for  $\psi = \neg\varphi$ , where  $\varphi$  is an LTL formula to be checked. An infinite path  $\pi$  of a model  $M$  that shows that  $M \models \mathbf{E}\psi$  is called a *witness* for  $\psi$ . The existence of a witness has to be demonstrated by exhibiting an appropriate path  $\pi_k$  of finite length  $k$ . In some cases, this can be done by finding a path  $\pi_k$  with a loop (two identical states); this can be expanded to an infinite path in which the loop is repeated infinitely often. But a finite path  $\pi_k$  can also demonstrate the existence of a witness, even if it is not known to have a loop. This can be understood in the framework of the truncated semantics as follows: The *bounded semantics without a loop* of [5] is the strong semantics for LTL formulas in positive normal form. If  $\pi_k$  satisfies  $\psi$  in this semantics, then by Theorems 7 and 3, every extension of  $\pi_k$  satisfies  $\psi$  in the neutral semantics. Assuming there is at least one transition from every state in  $M$ , there is an infinite extension  $\pi$  of  $\pi_k$  that is a computation path of  $M$ . Then  $\pi$  is a witness for  $\psi$ . Conversely, if there is no path of length  $k$  satisfying  $\psi$  in the bounded semantics without a loop, then every path of length  $k$  weakly satisfies  $\varphi$ . As noted in [5], the bounded semantics without a loop break the duality between strong and weak operators. The truncated semantics provide the missing dual weak semantics, and therefore render unnecessary the restriction of [5] to positive normal form.

The *completeness threshold* ( $\mathcal{CT}$ ) of [11] is reminiscent of our definitive prefix. However,  $\mathcal{CT}$  is defined with respect to a *model* and a formula while the definitive prefix is defined with respect to a *word* and a formula. Even if we try to compare the definitions by taking for a word  $w$  a model  $M_w$  that accepts  $w$  alone, the

definitions do not coincide: the definitive prefix for any word with respect to the formula  $Gp$  is  $\top$  but there exists a model  $M_w$  accepting  $Gp$  with a bounded  $CT$ .

In [17] the problem of determining the value of a formula over finite paths in simulation is also considered. Their semantics can be formulated using the notion of bad/good prefixes by defining a 3-valued satisfaction that returns *true* if a *good prefix* [12] is seen, *false* if a *bad prefix* is seen and *pending* otherwise. The resulting semantics is different than the truncated semantics and is quite similar to the abort semantics.

## 6 Conclusion and Future Work

We have considered the problem of reasoning in temporal logic over truncated as well as maximal paths, and have presented an elegant semantics for LTL augmented with a truncate operator over truncated and maximal paths. The semantics are defined relative to three views regarding the truth value of a formula when the truncation occurs before the evaluation of the formula completes. The *weak view* is consistent with a preference for false positives, the *strong view* with a preference for false negatives, and the *neutral view* with the desire to see as much evidence as can reasonably be expected from a finite path.

We have studied properties of the *truncated semantics* for the resulting logic  $LTL^{trunc}$ , as well as its relation to the *informative prefixes* of [12]. We have examined the relation between truncated paths and hardware resets, and have shown that our truncated semantics are mathematically equivalent to the *reset semantics* of [3].

Future work is to investigate how the weak/neutral/strong paradigm can be generalized: in particular, whether there are useful correspondences between alternative weak/neutral/strong semantics and other decision procedures for LTL, analogous to that between the truncated semantics and the classical tableau construction. Having a generalized framework, we might be able to find a logic that has the acceptable complexity of the truncated semantics, while allowing rewrite rules such as  $(\varphi \wedge \neg \varphi \stackrel{\text{def}}{=} \text{false})$ , which are prohibited in the truncated semantics.

In addition, we would like to combine the truncated semantics with those of  $LTL^\circ$  [9], to provide an integrated logic which supports both hardware clocks and hardware resets for both complete and incomplete verification methods.

## Acknowledgements

The IBM authors would like to thank Ishai Rabinovitz for many interesting and fruitful discussions, and Shoham Ben-David, Avigail Orni, and Sitvanit Ruah for close review and important comments on an earlier version of this work.

## References

1. Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs - automatic generation of simulation checkers from formal specifications. In *Proc. 12<sup>th</sup> International Conference on Computer Aided Verification (CAV)*, LNCS 1855, 2000.

2. R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Aborts vs resets in linear temporal logic. In *TACAS'03*, 2003. To appear.
3. R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *TACAS'02*, volume 2280 of *LNCS*. Springer, 2002.
4. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In *Proc. 10<sup>th</sup> International Conference on Computer Aided Verification (CAV)*, LNCS 1427, pages 184–194. Springer-Verlag, 1998.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5<sup>th</sup> International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 1579. Springer-Verlag, 1999.
6. L. Brouwer. *On the Foundations of Mathematics*. PhD thesis, Amsterdam, 1907. English translation in A. Heyting, Ed. L. E. J. Brouwer: Collected Works 1: Philosophy and Foundations of Mathematics, Amsterdam: North Holland / New York: American Elsevier (1975): 11-101.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead (Informatics'01)*, 2001.
8. C. Eisner and D. Fisman. Sugar 2.0 proposal presented to the Accellera Formal Verification Technical Committee, March 2002. At [http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar\\_2.0\\_Accellera.ps](http://www.haifa.il.ibm.com/projects/verification/sugar/Sugar_2.0_Accellera.ps).
9. C. Eisner, D. Fisman, J. Havlicek, A. McIsaac, and D. Van Campenhout. The definition of a temporal clock operator. 13th International Colloquium on Automata, Languages and Programming (ICALP), June 2003. To appear.
10. J. Havlicek, N. Levi, H. Miller, and K. Shultz. Extended CBV statement semantics, partial proposal presented to the Accellera Formal Verification Technical Committee, April 2002. At [http://www.eda.org/vfv/hm/att-0772/01-ecbv\\_statement\\_semantics.ps.gz](http://www.eda.org/vfv/hm/att-0772/01-ecbv_statement_semantics.ps.gz).
11. D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 298–309. Springer Verlag, January 2003.
12. O. Kupferman and M. Y. Vardi. Model checking of safety properties. In *Proc. 11<sup>th</sup> International Conference on Computer Aided Verification (CAV)*, LNCS 1633, 1999.
13. O. Lichtenstein. *Decidability, Completeness, and Extensions of Linear Time Temporal Logic*. PhD thesis, Weizmann Institute of Science, 1990.
14. O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Conf. on Logics of Programs*, LNCS 193, pages 196–218. Springer-Verlag, 1985.
15. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
16. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
17. J. Ruf, D. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on a multi-valued AR-automata. In *Proceedings of the DATE 2001 on Design, Automation and Test in Europe*, pages 742–748, March 2001.

# Structural Symbolic CTL Model Checking of Asynchronous Systems<sup>\*</sup>

Gianfranco Ciardo and Radu Siminiceanu

Department of Computer Science, College of William and Mary  
Williamsburg, VA 23187, USA  
{ciardo, radu}@cs.wm.edu

**Abstract.** In previous work, we showed how structural information can be used to efficiently generate the state-space of asynchronous systems. Here, we apply these ideas to symbolic CTL model checking. Thanks to a Kronecker encoding of the transition relation, we detect and exploit event locality and apply better fixed-point iteration strategies, resulting in orders-of-magnitude reductions for both execution times and memory consumption in comparison to well-established tools such as NuSMV.

## 1 Introduction

Verifying the correctness of a system, either by proving that it refines a specification or by determining that it satisfies certain properties, is an important step in system design. *Model checking* is concerned with the tasks of representing a system with an automaton, usually finite-state, and then showing that the initial state of this automaton satisfies a *temporal logic* statement [13].

Model checking has gained increasing attention since the development of techniques based on *binary decision diagrams (BDDs)* [4]. *Symbolic model checking* [6] is known to be effective for *computation tree logic (CTL)* [12], as it allows for the efficient storage and manipulation of the large sets of states corresponding to CTL formulae. However, practical limitations still exist. First, memory and time requirements might be excessive when tackling real systems. This is especially true since the size (in number of nodes) of the BDD encoding the set of states corresponding to a CTL formula is usually much larger *during* the fixed-point iterations than upon convergence. This has spurred work on distributed/parallel algorithms for BDD manipulation and on verification techniques that use only a fraction of the BDD nodes that would be required in principle [3, 19].

Second, symbolic model checking has been quite successful for hardware verification but software, in particular distributed software, has so far been considered beyond reach. This is because the state space of software is much larger, but also because of the widely-held belief that symbolic techniques work well only in synchronous settings. We attempt to dispel this myth by showing that symbolic model checking based on the model *structure* copes well with asynchronous be-

---

<sup>\*</sup> Work supported in part by the National Aeronautics and Space Administration under grants NAG-1-2168 and NAG-1-02095 and by the National Science Foundation under grants CCR-0219745 and ACI-0203971.

havior and even benefits from it. Furthermore, the techniques we introduce excel at reducing the *peak number of nodes* in the fixed-point iterations.

The present contribution is based on our earlier work in symbolic state-space generation using *multivalued decision diagrams (MDDs)*, *Kronecker encoding* of the next state function [17, 8], and the *saturation* algorithm [9]. This background is summarized in Section 2, which also discusses how to exploit the model structure for MDD manipulation. Section 3 contains our main contribution: improved computation of the basic CTL operators using structural model information. Section 4 gives memory and runtime results for our algorithms implemented in SMART [7] and compares them with NuSMV [11].

## 2 Exploiting the Structure of Asynchronous Models

We consider globally-asynchronous locally-synchronous systems specified by a tuple  $(\hat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{E}, \mathcal{N})$ , where the *potential state space*  $\hat{\mathcal{S}}$  is given by the product  $\mathcal{S}_K \times \dots \times \mathcal{S}_1$  of the  $K$  *local state spaces* of  $K$  *submodels*, i.e., a generic (global) state is  $\mathbf{i} = (i_K, \dots, i_1)$ ;  $\mathcal{S}^{init} \subseteq \hat{\mathcal{S}}$  is the set of *initial states*;  $\mathcal{E}$  is a set of (*asynchronous*) *events*; the *next-state function*  $\mathcal{N} : \hat{\mathcal{S}} \rightarrow 2^{\hat{\mathcal{S}}}$  is *disjunctively partitioned* [14] according to  $\mathcal{E}$ , i.e.,  $\mathcal{N} = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha$ , where  $\mathcal{N}_\alpha(\mathbf{i})$  is the set of states that can be reached when event  $\alpha$  *fires* in state  $\mathbf{i}$ ; we say that  $\alpha$  is *disabled* in  $\mathbf{i}$  if  $\mathcal{N}_\alpha(\mathbf{i}) = \emptyset$ .

With high-level models such as Petri nets or pseudo-code, the sets  $\mathcal{S}_k$ , for  $K \geq k \geq 1$ , might not be known a priori. Their derivation alongside the construction of the (*actual*) *state space*  $\mathcal{S} \subseteq \hat{\mathcal{S}}$ , defined by  $\mathcal{S} = \mathcal{S}^{init} \cup \mathcal{N}(\mathcal{S}^{init}) \cup \mathcal{N}^2(\mathcal{S}^{init}) \cup \dots = \mathcal{N}^*(\mathcal{S}^{init})$ , where  $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$ , is an interesting problem in itself [10]. Here, we assume that each  $\mathcal{S}_k$  is known and of finite size  $n_k$  and map its elements to  $\{0, \dots, n_k - 1\}$  for notational simplicity and efficiency.

Symbolic model checking manages subsets of  $\hat{\mathcal{S}}$  and relations over  $\hat{\mathcal{S}}$ . In the binary case, these are simply subsets of  $\{0, 1\}^K$  and of  $\{0, 1\}^{2K}$ , respectively, and are encoded as BDDs. Our structural approach instead uses MDDs to store sets and (boolean) sums of Kronecker matrix products to store relations. The use of MDDs has been proposed before [15], but their implementation through BDDs made them little more than a “user interface”. In [17], we showed instead that implementing MDDs directly may increase “locality”, thus the efficiency of state-space generation, if paired with our Kronecker encoding of  $\mathcal{N}$ . We use *quasi-reduced ordered* MDDs, directed acyclic edge-labeled multi-graphs where:

- Nodes are organized into  $K + 1$  *levels*. We write  $\langle k|p \rangle$  to denote a generic node, where  $k$  is the level and  $p$  is a unique index for a node at that level.
- Level  $K$  contains only a single *non-terminal* node  $\langle K|r \rangle$ , the *root*, whereas levels  $K - 1$  through 1 contain one or more non-terminal nodes.
- Level 0 consists of the two *terminal* nodes,  $\langle 0|0 \rangle$  and  $\langle 0|1 \rangle$ .
- A non-terminal node  $\langle k|p \rangle$  has  $n_k$  arcs, labeled from 0 to  $n_k - 1$ , pointing to nodes at level  $k - 1$ . If the arc labeled  $i_k$  points to node  $\langle k - 1|q \rangle$ , we write  $\langle k|p \rangle[i_k] = q$ . *Duplicate* nodes are not allowed but, unlike the (*strictly*)

*reduced* ordered decision diagrams of [15], *redundant* nodes where all arcs point to the same node are allowed (both versions are canonical [16]).

Let  $\mathcal{A}(\langle k|p \rangle)$  be the set of tuples  $(i_K, \dots, i_{k+1})$  labeling paths from  $\langle K|r \rangle$  to node  $\langle k|p \rangle$ , and  $\mathcal{B}(\langle k|p \rangle)$  the set of tuples  $(i_k, \dots, i_1)$  labeling paths from  $\langle k|p \rangle$  to  $\langle 0|1 \rangle$ . In particular,  $\mathcal{B}(\langle K|r \rangle)$  and  $\mathcal{A}(\langle 0|1 \rangle)$  specify the states encoded by the MDD.

A more drastic departure from traditional symbolic approaches is our encoding of  $\mathcal{N}$  [17], inspired by the representation of the transition rate matrix for a continuous-time Markov chain by means of a (real) sum of Kronecker products [5, 18]. This requires a *Kronecker-consistent* decomposition of the model into submodels, i.e., there must exist functions  $\mathcal{N}_{\alpha,k} : \mathcal{S}_k \rightarrow 2^{\mathcal{S}_k}$ , for  $\alpha \in \mathcal{E}$  and  $K \geq k \geq 1$ , such that, for any  $\mathbf{i} \in \hat{\mathcal{S}}$ ,  $\mathcal{N}_{\alpha}(\mathbf{i}) = \mathcal{N}_{\alpha,K}(i_K) \times \dots \times \mathcal{N}_{\alpha,1}(i_1)$ .

$\mathcal{N}_{\alpha,k}(i_k)$  represents the set of local states locally reachable (i.e., for submodel  $k$  in isolation) from local state  $i_k$  when  $\alpha$  fires. In particular,  $\alpha$  is disabled in any global state whose  $k^{\text{th}}$  component  $i_k$  satisfies  $\mathcal{N}_{\alpha,k}(i_k) = \emptyset$ . This consistency requirement is quite natural for asynchronous systems. Indeed, it is always satisfied by formalisms such as Petri nets, for which any partition of the  $P$  places of the net into  $K \leq P$  subsets is consistent. We define the boolean incidence matrices  $\mathbf{W}_{\alpha,k} \in \{0,1\}^{\mathcal{S}_k \times \mathcal{S}_k}$  so that  $\mathbf{W}_{\alpha,k}[i_k, j_k] = 1$  iff  $j_k \in \mathcal{N}_{\alpha,k}(i_k)$ . Then,  $\mathbf{W} = \bigvee_{\alpha \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{W}_{\alpha,k}$  encodes  $\mathcal{N}$ , i.e.,  $\mathbf{W}[\mathbf{i}, \mathbf{j}] = 1$  iff  $\mathbf{j} \in \mathcal{N}(\mathbf{i})$ , where  $\bigotimes$  denotes the Kronecker product of matrices, and the mixed-base value  $\sum_{k=1}^K i_k \prod_{l=1}^{k-1} n_l$  of  $\mathbf{i}$  is used when indexing  $\mathbf{W}$ .

## 2.1 Locality, In-Place-Updates, and Saturation

One important advantage of our Kronecker encoding is its ability to evidence the *locality* of events inherently present in most asynchronous systems. We say that an event  $\alpha$  is *independent* of level  $k$  if  $\mathbf{W}_{\alpha,k}$  is the identity matrix; this means that the  $k^{\text{th}}$  local state does not affect the enabling of  $\alpha$ , nor is modified by the firing of  $\alpha$ . We then define  $\text{Top}(\alpha)$  and  $\text{Bot}(\alpha)$  to be the maximum and minimum levels on which  $\alpha$  depends. Since an event must be able to modify at least some local state to be meaningful, we can assume that these levels are always well defined, i.e.,  $K \geq \text{Top}(\alpha) \geq \text{Bot}(\alpha) \geq 1$ .

One advantage of our encoding is that, for practical asynchronous systems, most  $\mathbf{W}_{\alpha,k}$  are the identity matrix (thus do not need to be stored explicitly) while the rest usually have very few nonzero entries per row (thus can be stored with sparse data structures). This is much more compact than the BDD or MDD storage of  $\mathcal{N}$ . In a BDD representation of  $\mathcal{N}_{\alpha}$ , for example, an edge skipping levels  $k$  and  $k'$  (the  $k^{\text{th}}$  components of the “from” and “to” states, respectively) means that, after  $\alpha$  fires, the  $k^{\text{th}}$  component can be either 0 or 1, regardless of whether it was 0 or 1 before the firing. The more common behavior is instead the one where 0 remains 0 and 1 remains 1, the default in our Kronecker encoding.

In addition to reducing the memory requirements to encode  $\mathcal{N}$ , the Kronecker encoding allows us to exploit locality to reduce the execution time when generating the state space. In [17], we performed an iteration of the form

```

repeat
  for each  $\alpha \in \mathcal{E}$  do  $\boxed{\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{N}_\alpha(\mathcal{S})}$ 
until  $\mathcal{S}$  does not change

```

with  $\mathcal{S}$  initialized to  $\mathcal{S}^{init}$ . If  $Top(\alpha) = k$ ,  $Bot(\alpha) = l$ , and  $\mathbf{i} \in \mathcal{S}$ , then, for any  $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$  we have  $\mathbf{j} = (i_K, \dots, i_{k+1}, j_k, \dots, j_l, i_{l-1}, \dots, i_1)$ . Thus, we descend from the root of the MDD encoding the current  $\mathcal{S}$  and, only when encountering a node  $\langle k|p \rangle$  we call the recursive function  $Fire(\alpha, \langle k|p \rangle)$  to compute the resulting node at the same level  $k$  using the information encoded by  $\mathbf{W}_{\alpha,k}$ ; furthermore, after processing a node  $\langle l|q \rangle$ , with  $l = Bot(\alpha)$ , the recursive  $Fire$  calls stop.

In [8], we gained further efficiency by performing *in-place updates* of (some) MDD nodes. This is based on the observation that, for any other  $\mathbf{i}' \in \mathcal{S}$  whose last  $k$  components coincide with those of  $\mathbf{i}$  and whose first  $K - k$  components  $(i'_K, \dots, i'_{k+1})$  lead to the same node  $\langle k|p \rangle$  as  $(i_K, \dots, i_{k+1})$ , we can immediately conclude that  $\mathbf{j}' = (i'_K, \dots, i'_{k+1}, j_k, \dots, j_l, i_{l-1}, \dots, i_1)$  is also reachable. Thus, we performed an iteration of the form (let  $\mathcal{E}_k = \{\alpha : Top(\alpha) = k\}$ )

```

repeat
  for  $k = 1$  to  $K$  do
    for each node  $\langle k|p \rangle$  do
      for each  $\alpha \in \mathcal{E}_k$  do  $\boxed{\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{A}(\langle k|p \rangle) \times Fire(\alpha, \langle k|p \rangle)}$ 
until  $\mathcal{S}$  does not change

```

where the “ $\mathcal{A}(\langle k|p \rangle) \times$ ” operation comes at no cost, since it is implied by starting the firing of  $\alpha$  “in the middle of the MDD” and directly updating node  $\langle k|p \rangle$ .

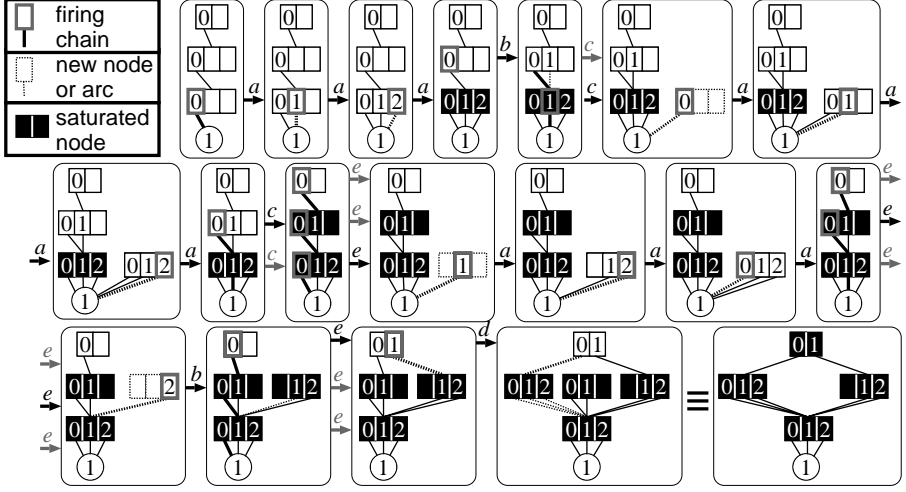
The memory and time savings due to in-place updates are compounded to those due to locality. Especially when studying asynchronous systems with “tall” MDDs (large  $K$ ), this results in orders-of-magnitude improvements with respect to traditional symbolic approaches. However, even greater savings are achieved by *saturation* [9], a new iteration control strategy made possible by the use of structural model information. A node  $\langle k|p \rangle$  is *saturated* if it is a fixed point with respect to firing any event that is independent of all levels above  $k$ :

$$\forall l, k \geq l \geq 1, \forall \alpha \in \mathcal{E}_l, \mathcal{A}(\langle k|p \rangle) \times \mathcal{B}(\langle k|p \rangle) \supseteq \mathcal{N}_\alpha(\mathcal{A}(\langle k|p \rangle) \times \mathcal{B}(\langle k|p \rangle)).$$

With saturation, the traditional global fixed-point iteration for the overall MDD disappears. Instead, we start saturating the node at level 1 (assuming  $|\mathcal{S}^{init}| = 1$ , the initial MDD contains one node per level), move up in the MDD saturating nodes, and end the process when we have saturated the root. To saturate a node  $\langle k|p \rangle$ , we exhaustively fire each event  $\alpha \in \mathcal{E}_k$  in it, using in-place updates at level  $k$ . Each required  $Fire(\alpha, \langle k|p \rangle)$  call may create nodes at lower levels, which are recursively saturated before completing the  $Fire$  call itself (see Fig. 1).

Saturation has numerous advantages over traditional methods, resulting in enormous memory and time savings. Once  $\langle k|p \rangle$  is saturated, we never fire an event  $\alpha \in \mathcal{E}_k$  in it again. Only saturated nodes appear in the unique table and operation caches. Finally, most of these nodes will still be present in the final MDD (non-saturated nodes are *guaranteed* not to be part of it). In fact, the peak and final number of nodes differ by a mere constant in some models [9].

$$\begin{aligned}
W_{a,3} &= \mathbf{I} & W_{b,3} &= \mathbf{I} & W_{c,3} &= \mathbf{I} & W_{d,3} &= \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} & W_{e,3} &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \\
W_{a,2} &= \mathbf{I} & W_{b,2} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} & W_{c,2} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & W_{d,2} &= \mathbf{I} & W_{e,2} &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
W_{a,1} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} & W_{b,1} &= \mathbf{I} & W_{c,1} &= \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} & W_{d,1} &= \mathbf{I} & W_{e,1} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned}$$



**Fig. 1:** Encoding of  $\mathcal{N}$  and generation of  $\mathcal{S}$  using saturation:  $K = 3$ ,  $\mathcal{E} = \{a, b, c, d, e\}$ . Arrows between frames indicate the event being fired (a darker shade is used for the event label on the “active” level in the firing chain). The firing sequence is:  $a$  (3 times),  $b$ ,  $c$  (at level 1),  $a$  (interrupting  $c$ , 3 times to saturate a new node),  $c$  (resumed, at level 2),  $e$  (at level 1),  $a$  (interrupting  $e$ , 3 times to saturate a new node),  $e$  (resumed, at level 2),  $b$  (interrupting  $e$ ),  $e$  (resumed, at level 3), and finally  $d$  (the union of  $\begin{bmatrix} 0 & 1 \end{bmatrix}$  and  $\begin{bmatrix} 1 & 2 \end{bmatrix}$  at level 2, i.e.,  $\begin{bmatrix} 0 & 1 & 2 \end{bmatrix}$ , is saturated by definition). There is at most one unsaturated node per level, and the one at the lowest level is being saturated.

### 3 Structural-Based CTL Model Checking

After having summarized the distinguishing features of the data structures and algorithms we employ for state-space generation, we now consider how to apply them to symbolic model checking. CTL [12] is widely used due to its simple yet expressive syntax and to the existence of efficient algorithms for its analysis [6]. In CTL, operators occur in pairs: the path quantifier, either  $A$  (on all paths) or  $E$  (there exists a path), is followed by the tense operator, one of  $X$  (next),  $F$  (future, or finally),  $G$  (globally, or generally), and  $U$  (until). Of the eight possible pairings, only a *generator* (sub)set needs to be implemented in a model checker, as the remaining operators can be expressed in terms of those in the set [13].  $\{EX, EU, EG\}$  is such a set, but the following discusses also  $EF$  for clarity.



### 3.1 The $EX$ Operator

**Semantics:**  $\mathbf{i}^0 \models EXp$  iff  $\exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0)$  s.t.  $\mathbf{i}^1 \models p$ . (“ $\models$ ” means “satisfies”)

In our notation,  $EX$  corresponds to the inverse function of  $\mathcal{N}$ , the previous-state function,  $\mathcal{N}^{-1}$ . With our Kronecker matrix encoding, the inverse of  $\mathcal{N}_\alpha$  is simply obtained by transposing the incidence matrices  $\mathbf{W}_{\alpha,k}$  in the Kronecker product, thus  $\mathcal{N}^{-1}$  is encoded as  $\bigvee_{\alpha \in \mathcal{E}} \bigotimes_{K \geq k \geq 1} \mathbf{W}_{\alpha,k}^T$ .

To compute the set of states where  $EXp$  is satisfied, we can follow the same idea used to fire events in an MDD node during our state-space generation: given the set  $\mathcal{P}$  of states satisfying formula  $p$ , we can accumulate the effect of “firing backward” each event by taking advantage of locality and in-place updates. This results in an efficient calculation of  $EX$ . Computing its reflexive and transitive closure, that is, the backward reachability operator  $EF$ , is a much more difficult challenge, which we consider next.

### 3.2 The $EF$ Operator

**Semantics:**  $\mathbf{i}^0 \models EFp$  iff  $\exists n \geq 0, \exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0), \dots, \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$  s.t.  $\mathbf{i}^n \models p$ .

In our approach, the construction of the set of states satisfying  $EFp$  is analogous to the saturation algorithm for state-space generation, with two differences. Besides using the transposed incidence matrices  $\mathbf{W}_{\alpha,k}^T$ , the execution starts with the set  $\mathcal{P}$ , not a single state. These differences do not affect the applicability of saturation, which retains all its substantial time and memory benefits.

### 3.3 The $EU$ Operator

**Semantics:**  $\mathbf{i}^0 \models E[pUq]$  iff  $\exists n \geq 0, \exists \mathbf{i}^1 \in \mathcal{N}(\mathbf{i}^0), \dots, \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$  s.t.  $\mathbf{i}^n \models q$  and  $\mathbf{i}^m \models p$  for all  $m < n$ . (in particular,  $\mathbf{i} \models q$  implies  $\mathbf{i} \models E[pUq]$ )

The traditional computation of the set of states satisfying  $E[pUq]$  uses a least fixed point algorithm. Starting with the set  $\mathcal{Q}$  of states satisfying  $q$ , it iteratively adds all the states that reach them on paths where property  $p$  holds (see Algorithm *EUtrad* in Fig. 2). The number of iterations to reach the fixed point is  $\max_{\mathbf{i} \in \mathcal{S}} (\min \{n \mid \exists \mathbf{i}^0 \in \mathcal{Q} \wedge \forall 0 < m \leq n, \exists \mathbf{i}^m \in \mathcal{N}^{-1}(\mathbf{i}^{m-1}) \cap \mathcal{P} \wedge \mathbf{i} = \mathbf{i}^n\})$ .

*EUtrad*(in  $\mathcal{P}, \mathcal{Q}$  : set of state) : set of state

1. declare  $\mathcal{X}, \mathcal{Y}$  : set of state;
2.  $\mathcal{X} \leftarrow \mathcal{Q}$ ; • initialize  $\mathcal{X}$  with all states in  $\mathcal{Q}$
3. repeat
4.    $\mathcal{Y} \leftarrow \mathcal{X}$ ;
5.    $\mathcal{X} \leftarrow \mathcal{X} \cup (\mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P})$ ; • add predecessors of states in  $\mathcal{X}$  that are in  $\mathcal{P}$
6. until  $\mathcal{Y} = \mathcal{X}$ ;
7. return  $\mathcal{X}$ ;

**Fig. 2:** Traditional algorithm to compute the set of states satisfying  $E[pUq]$ .

**Applying Saturation to *EU*.** As the main contribution of this paper, we propose a new approach to computing *EU* based on saturation. The challenge in applying saturation arises from the need to “filter out” states not in  $\mathcal{P}$  (line 5 of Algorithm *EUtrad*): as soon as a new predecessor of the working set  $\mathcal{X}$  is obtained, it must be intersected with  $\mathcal{P}$ . Failure to do so can result in paths to  $\mathcal{Q}$  that stray, even temporarily, out of  $\mathcal{P}$ . However, saturation works in a highly localized manner, adding states out of breadth-first-search (BFS) order. Performing an expensive intersection after each firing would add enormous overhead, since our firings are very lightweight operations. To cope with this problem, we propose a “partial” saturation that is applied to a subset of events for which no filtering is needed. These are the events whose firing is *guaranteed* to preserve the validity of the formula  $p$ . For the remaining events, BFS with filtration must be used. The resulting global fixed point iteration interleaves these two phases (see Fig. 3). The following classification of events is analogous to, but different from, the *visible* vs. *invisible* one proposed for partial order reduction [1].

**Definition 1** In a discrete state model  $(\hat{\mathcal{S}}, \mathcal{S}^{init}, \mathcal{E}, \mathcal{N})$ , an event  $\alpha$  is *dead* with respect to a set of states  $\mathcal{X}$  if there is no state in  $\mathcal{X}$  from which its firing leads to a state in  $\mathcal{X}$ , i.e.,  $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \cap \mathcal{X} = \emptyset$  (this includes the case where  $\alpha$  is always disabled in  $\mathcal{X}$ ); it is *safe* if it is not dead and its firing cannot lead from a state not in  $\mathcal{X}$  to a state in  $\mathcal{X}$ , i.e.,  $\emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{X}) \subseteq \mathcal{X}$ ; it is *unsafe* otherwise, i.e.,  $\mathcal{N}_\alpha^{-1}(\mathcal{X}) \setminus \mathcal{X} \neq \emptyset \wedge \mathcal{N}_\alpha^{-1}(\mathcal{X}) \cap \mathcal{X} \neq \emptyset$ .  $\square$

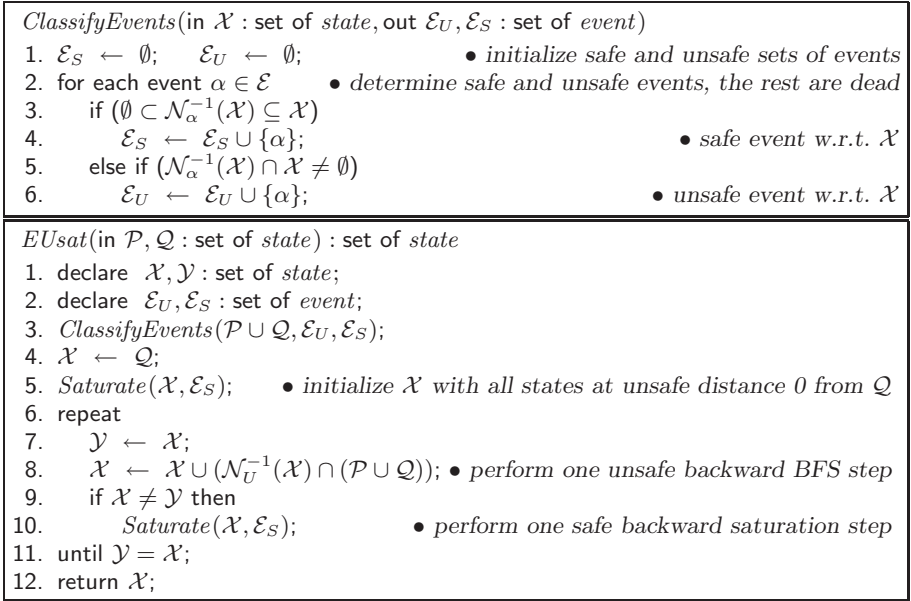
Given a formula  $E[pUq]$ , we first classify the safety of events through static analysis. Then, each *EU* fixed point iteration consists of two backward steps: BFS on unsafe events followed by saturation on safe events. Since saturation is in turn a fixed point computation, the resulting algorithm computes a nested fixed point. Note that the operators used in both steps are monotonic (the working set  $\mathcal{X}$  is increasing), a condition for applying saturation and in-place updates.

**Note 1** Dead events can be ignored altogether by our *EU<sub>sat</sub>* algorithm, since the working set  $\mathcal{X}$  is always a subset of  $\mathcal{P} \cup \mathcal{Q}$ .

**Note 2** The *Saturate* procedure in line 10 of *EU<sub>sat</sub>* is analogous to the one we use for *EF*, except that it is restricted to a subset  $\mathcal{E}_S$  of events.

**Note 3** *ClassifyEvents* has the same time complexity as one *EX* step and is called only once prior to the fixed point iterations.

**Note 4** To simplify the description of *EU<sub>sat</sub>*, we call *ClassifyEvents* with the filter  $\mathcal{P} \cup \mathcal{Q}$ , i.e.,  $\mathcal{E}_S = \{\alpha : \emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{P} \cup \mathcal{Q}) \subseteq \mathcal{P} \cup \mathcal{Q}\}$ . With a slightly more complex initialization in *EU<sub>sat</sub>*, we could use instead the smaller filter  $\mathcal{P}$ , i.e.,  $\mathcal{E}_S = \{\alpha : \emptyset \subset \mathcal{N}_\alpha^{-1}(\mathcal{P}) \subseteq \mathcal{P}\}$ . In practice, both sets of events could be computed. Then, if one is a strict superset of the other, it should be used, since the larger  $\mathcal{E}_S$  is, the more *EU<sub>sat</sub>* behaves like our efficient *EF* saturation; otherwise, some heuristic must be used to choose between the two.



**Fig. 3:** Saturation-based algorithm to compute the set of states satisfying  $E[pUq]$ .

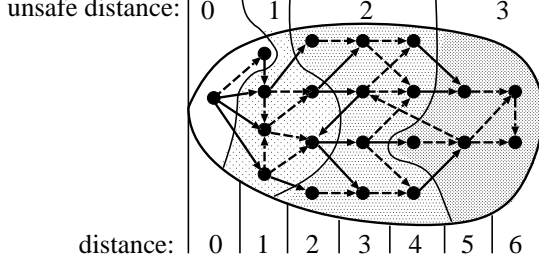
**Note 5** The number of *EUsat* iterations is 1 plus the “unsafe distance from  $\mathcal{P}$  to  $\mathcal{Q}$ ”,  $\max_{\mathbf{i} \in \mathcal{P}} (\min \{n | \exists \mathbf{i}^0 \in \mathcal{R}_S^*(\mathcal{Q}) \wedge \forall 0 < m \leq n, \exists \mathbf{j}^m \in \mathcal{R}_U^*(\mathcal{R}_U(\mathbf{i}^{m-1}) \cap \mathcal{P}) \wedge \mathbf{i} = \mathbf{i}^n\})$ , where  $\mathcal{R}_S(\mathcal{X}) = \bigcup_{\alpha \in \mathcal{E}_S} \mathcal{N}_\alpha^{-1}(\mathcal{X})$  and  $\mathcal{R}_U(\mathcal{X}) = \bigcup_{\alpha \in \mathcal{E}_U} \mathcal{N}_\alpha^{-1}(\mathcal{X})$  are the sets of “safe predecessors” and “unsafe predecessors” of  $\mathcal{X}$ , respectively.

**Lemma 1** Iteration  $d$  of *EUsat* finds all states  $\mathbf{i}$  at unsafe distance  $d$  from  $\mathcal{Q}$ .

**Proof.** By induction on  $d$ . Base:  $d = 0 \Rightarrow \mathbf{i} \in \mathcal{R}_S^*(\mathcal{Q})$  which is a subset of  $\mathcal{X}$  (lines 4,5). Inductive step: suppose all states at unsafe distance  $m \leq d$  are added to  $\mathcal{X}$  in the  $m^{\text{th}}$  iteration. By definition, a state  $\mathbf{i}$  at unsafe distance  $d+1$  satisfies:  $\exists \mathbf{i}^0 \in \mathcal{R}_S^*(\mathcal{Q}) \wedge \forall 0 < m \leq d+1, \exists \mathbf{j}^m \in \mathcal{R}_U(\mathbf{i}^{m-1}) \cap \mathcal{P}, \exists \mathbf{i}^m \in \mathcal{R}_S^*(\mathbf{j}^m)$ , and  $\mathbf{i} = \mathbf{i}^{d+1}$ . Then,  $\mathbf{i}^m$  and  $\mathbf{j}^m$  are at unsafe distance  $m$ . By the induction hypothesis, they are added to  $\mathcal{X}$  in iteration  $m$ . In particular,  $\mathbf{i}^d$  is a new state found in iteration  $d$ . This implies that the algorithm must execute another iteration, which finds  $\mathbf{j}^{d+1}$  as an unsafe predecessor of  $\mathbf{i}^d$  (line 8). Since  $\mathbf{i}$  is either  $\mathbf{j}^{d+1}$  or can reach it through safe events alone, it is added to  $\mathcal{X}$  (line 10).  $\square$

**Theorem 1** Algorithm *EUsat* returns the set  $\mathcal{X}$  of states satisfying  $E[pUq]$ .

**Proof.** It is immediate to see that *EUsat* terminates, since its working set is a monotonically increasing subset of  $\hat{\mathcal{S}}$ , which is finite. Let  $\mathcal{Y}$  be the set of states satisfying  $E[pUq]$ . We have (i)  $\mathcal{Q} \subseteq \mathcal{X}$  (line 4) (ii) every state in  $\mathcal{X}$  can reach a state in  $\mathcal{Q}$  through a path in  $\mathcal{X}$ , and (iii)  $\mathcal{X} \subseteq \mathcal{P} \cup \mathcal{Q}$  (lines 8,10). This implies



**Fig. 4:** Comparing BFS and saturation order: distance vs. unsafe distance.

$\mathcal{X} \subseteq \mathcal{Y}$ . Since any state in  $\mathcal{Y}$  is at some finite unsafe distance  $d$  from  $\mathcal{Q}$ , by Lemma 1 we conclude that  $\mathcal{Y} \subseteq \mathcal{X}$ . The two set inclusions imply  $\mathcal{X} = \mathcal{Y}$ .  $\square$

Figure 4 illustrates the way our exploration differs from BFS. Solid and dashed arcs represent unsafe and safe transitions, respectively. The shaded areas encircle the explored regions after each iteration of *EUsat*, four in this case. *EUtrad* would instead require seven iterations to explore the entire graph (states are aligned vertically according to their BFS depth).

**Note 6** Our approach exhibits “graceful degradation”. In the best case, all events are safe, and *EUsat* performs just one saturation step and stops. This happens for example when  $p \vee q \equiv \text{true}$ , which includes the special case  $p \equiv \text{true}$ . As  $E[\text{true} \cup q] \equiv EFq$ , we simply perform backward reachability from  $\mathcal{Q}$  using saturation on the entire set of events. In the worst case, all events are unsafe, and *EUsat* performs the same steps as *EUtrad*. But even then, locality and our Kroecker encoding can still substantially improve the efficiency of the algorithm.

### 3.4 The *EG* Operator

**Semantics:**  $\mathbf{i}^0 \models EGp$  iff  $\forall n > 0, \exists \mathbf{i}^n \in \mathcal{N}(\mathbf{i}^{n-1})$  s.t.  $\mathbf{i}^n \models p$ .

In graph terms, consider the reachability subgraph obtained by restricting the transition relation to states in  $\mathcal{P}$ . Then, *EGp* holds in any state belonging to, or reaching, a nontrivial strongly connected component (SCC) of this subgraph.

Algorithm *EGtrad* in Fig. 5 shows the traditional greatest fixed point iteration. It initializes the working set  $\mathcal{X}$  with all states in  $\mathcal{P}$  and gradually eliminates states that have no successor in  $\mathcal{X}$  until only the SCCs of  $\mathcal{P}$  and their incoming paths along states in  $\mathcal{P}$  are left. The number of iterations equals the maximum length of any path over  $\mathcal{P}$  that does *not* lead to such an SCC.

**Applying Saturation to *EG*.** *EGtrad* is a greatest fixed point, so to speed it up we must *eliminate* unwanted states faster. The criterion for a state  $\mathbf{i}$  is a *conjunction*:  $\mathbf{i}$  should be eliminated if *all* its successors are not in  $\mathcal{P}$ . Since it considers a single event at a time and makes local decisions that must be globally correct, it would appear that saturation cannot be used to improve *EGtrad*.

<pre> EGtrad(in <math>\mathcal{P}</math> : set of state) : set of state 1. declare <math>\mathcal{X}, \mathcal{Y}</math> : set of state; 2. <math>\mathcal{X} \leftarrow \mathcal{P}</math>; 3. repeat 4.   <math>\mathcal{Y} \leftarrow \mathcal{X}</math>; 5.   <math>\mathcal{X} \leftarrow \mathcal{N}^{-1}(\mathcal{X}) \cap \mathcal{P}</math>; 6. until <math>\mathcal{Y} = \mathcal{X}</math>; 7. return <math>\mathcal{X}</math>; </pre>	<pre> EGsat(in <math>\mathcal{P}</math> : set of state) : set of state 1. declare <math>\mathcal{X}, \mathcal{Y}, \mathcal{C}, \mathcal{T}</math> : set of state; 2. <math>\mathcal{C} \leftarrow EU_{sat}(\mathcal{P}, \{\mathbf{i} \in \mathcal{P} : \mathbf{i} \in \mathcal{N}(\mathbf{i})\})</math>; 3. <math>\mathcal{T} \leftarrow \emptyset</math>; 4. while <math>\exists \mathbf{i} \in \mathcal{P} \setminus (\mathcal{C} \cup \mathcal{T})</math> do 5.   <math>\mathcal{X} \leftarrow EU_{sat}(\mathcal{P} \setminus \mathcal{C}, \{\mathbf{i}\})</math>; 6.   <math>\mathcal{Y} \leftarrow ES_{sat}(\mathcal{P} \setminus \mathcal{C}, \{\mathbf{i}\})</math>; 7.   if <math> \mathcal{X} \cap \mathcal{Y}  &gt; 1</math> then 8.     <math>\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{X}</math>; 9.   else 10.    <math>\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathbf{i}\}</math>; 11. return <math>\mathcal{C}</math>; </pre>
---	--

**Fig. 5:** Traditional and saturation-based *EG* algorithms.

However, Fig. 5 shows an algorithm for *EG* which, like [2, 20], enumerates the SCCs by finding forward and backward reachable sets from a state. However, it uses saturation, instead of breadth-first search. In line 2, Algorithm *EGsat* disposes of selfloop states in  $\mathcal{P}$  and of the states reaching them through paths in  $\mathcal{P}$  (selfloops can be found by performing *EX* using a modified set of matrices  $\mathbf{W}_{\alpha,k}$  where off-diagonal entries are set to zero). Then, it chooses a single state  $\mathbf{i} \in \mathcal{P}$  and builds the backward and forward reachable sets from  $\mathbf{i}$  restricted to  $\mathcal{P}$ , using *EU<sub>sat</sub>* and *ES<sub>sat</sub>* (*ES* is the dual in the past of *EU*; it differs from *EU<sub>sat</sub>* only in that it does not transpose the matrices  $\mathbf{W}_{\alpha,k}$ ). If  $\mathcal{X}$  and  $\mathcal{Y}$  have more than just  $\mathbf{i}$  in common,  $\mathbf{i}$  belongs to a nontrivial SCC and all of  $\mathcal{X}$  is part of our answer  $\mathcal{C}$ . Otherwise, we add  $\mathbf{i}$  to the set  $\mathcal{T}$  of trivial SCCs ( $\mathbf{i}$  might nevertheless reach a nontrivial SCC, in  $\mathcal{Y}$ , but we have no easy way to tell). The process ends when  $\mathcal{P}$  has been partitioned into  $\mathcal{C}$ , containing nontrivial SCCs and states reaching them over  $\mathcal{P}$ , and  $\mathcal{T}$ , containing trivial SCCs.

*EGsat* is more efficient than our *EGtrad* only in special cases. An example is when the *EU<sub>sat</sub>* and *ES<sub>sat</sub>* calls in *EGsat* find each next state on a long path of trivial SCCs through a single lightweight firing, while *EGtrad* always attempts firing each event at each iteration. In the worst case, however, *EGsat* can be much worse than not only *EGtrad*, but even an explicit approach. For this reason, the next section discusses only *EGtrad*, which is guaranteed to benefit from locality and the Kronecker encoding.

## 4 Results

We implemented our algorithms in SMART [7] and compared them with NuSMV (version 2.1.2), on a 2.2 GHz Pentium IV Linux workstation with 1GB of RAM. Our examples are chosen from the world of distributed systems and protocols. Each system is modeled in the SMART and NuSMV input languages. We verified that the two models are equivalent, by checking that they have the same sets of potential and reachable states and the same transition relation.

We briefly describe the chosen models and their characteristics. Detailed descriptions can be found in [7]. The randomized asynchronous leader election protocol solves the problem of designating a unique leader among  $N$  participants by sending messages along a unidirectional ring. The dining philosophers and the round robin protocol models solve a specific type of mutual exclusion problem among  $N$  processes. The slotted ring models a communication protocol in a network of  $N$  nodes. The flexible manufacturing system model describes a factory with three production units where  $N$  parts of each of three different types move around on pallets (for compatibility with NuSMV, we had to change immediate events in the original SMART model [7] into timed ones). This is the only model where the number of levels in the MDD is fixed, not depending on  $N$  (of course, the size of the local state spaces  $\mathcal{S}_k$ , depends instead on  $N$ ). All of these models are characterized by loose connectivity between components, i.e., they are examples of globally-asynchronous locally-synchronous systems. We used the best known variable ordering for SMART and NuSMV (they coincide in all models except round robin, where, for best performance, NuSMV uses the reverse of the one for SMART). The time to build the encoding of  $\mathcal{N}$  is not included in the table; while this time is negligible for our Kronecker encoding, it can be quite substantial for NuSMV, at times exceeding the reported runtimes.

Table 4 shows the state-space size, runtime (sec), and peak memory consumption (MB) for the five models, counting MDD nodes plus Kronecker matrices in SMART, and BDD nodes in NuSMV. There are three sets of columns: state-space generation (analogous to *EF*), *EU*, and *EG*. See [7] for the meaning of the atomic propositions. In NuSMV, it is possible to evaluate *EU* expressions without explicitly building the state space first; however, this substantially increases the runtime, so that it almost equals the sum of the state-space generation and *EU* entries. The same holds for *EG*. In SMART the state-space construction is always executed in advance, hence the memory consumption includes the MDD for the state space. We show the largest parameter for which NuSMV can build the state space in the penultimate row of each model, while the last row shows the largest parameter for which SMART can evaluate the *EU* and *EG*.

Overall, SMART outperforms NuSMV time- and memory-wise. Saturation excels at state-space generation, with improvements exceeding 100,000 in time and 1,000 in memory. Indeed, SMART can scale  $N$  even more than shown, e.g., 10 for the leader election, 10,000 for philosophers, 200 for round robin, and 150 for FMS. For *EU*, we can see the effect of the data structures and of the algorithm separately, since we report for both *EU<sub>trad</sub>* and *EU<sub>sat</sub>*. When comparing the two, the latter reaches a fixed point in fewer iterations (recall Fig. 4) and uses less memory. While each *EU<sub>sat</sub>* iteration is more complex, it also operates on smaller MDDs, one of the benefits of saturation. The performance gain is more evident in large models, where *EU<sub>trad</sub>* runs out of memory before completing the task and is up to 20 times slower. The comparison between our *EU<sub>trad</sub>*, *EG<sub>trad</sub>* and NuSMV highlights instead the differences between data structures. SMART is still faster and uses much less memory, suggesting that the Kronecker representation for the transition relation is much more efficient than the  $2K$ -level BDD representation.

N	S	State-space generation						EU query						EG query										
		NuSMV			SMART			NuSMV			SMART			NuSMV			SMART							
		time	mem	time	mem	time	mem	after SS	alone	time	mem	iter	time	mem	time	mem	time	mem	time	mem	time	mem		
Leader: $K = 2N$ , $ \mathcal{E}  = N^2 + 13N$																								
38.49×10 <sup>2</sup>		0.1	2	0.04	<.5	0.1	3	18.3	12	43	0.02	<.5	22	0.02	<.5	4	0.7	4	0.02	<.5	4	0.02	<.5	
41.15×10 <sup>4</sup>		2.1	10	0.27	<.5	2.3	11	8104.7	371	62	0.36	1	38	0.27	1	232.8	12	1189.1	235	0.11	2	0.11	2	
51.50×10 <sup>5</sup>		56.0	29	1.49	1	52.0	33	—	—	81	3.74	7	52	3.09	7	18023.6	104	—	—	0.44	9	0.44	9	
61.89×10 <sup>6</sup>		1063.7	295	7.35	3	—	—	—	—	101	46.90	30	66	35.67	28	—	—	—	—	1.64	38	1.64	38	
72.39×10 <sup>7</sup>		—	—	40.64	7	—	—	—	—	121	690.85	116	85	416.85	101	—	—	—	—	7.15	128	7.15	128	
Philosophers: $K = \lceil N/2 \rceil$ , $ \mathcal{E}  = 4N$																								
203.46×10 <sup>12</sup>		0.8	6	0.02	<.5	0.1	7	0.8	6	40	0.03	<.5	4	0.02	<.5	0.1	8	1.1	6	0.01	<.5	6	0.01	<.5
502.23×10 <sup>31</sup>		36.0	46	0.07	<.5	1.2	46	39.7	46	100	0.17	1	4	0.06	1	0.9	46	132.3	50	0.02	1	0.02	1	
1004.96×10 <sup>62</sup>		1134.8	316	0.15	<.5	7.9	316	1121.8	316	200	0.67	3	4	0.14	3	9.0	316	2525.3	358	0.05	3	0.05	3	
5003.03×10 <sup>313</sup>		—	—	1.01	1	—	—	—	—	1000	19.09	78	4	0.77	60	—	—	—	—	0.28	58	0.28	58	
Slotted ring: $K = N$ , $ \mathcal{E}  = 8N$																								
55.38×10 <sup>5</sup>		0.1	1	<.005	<.5	0.0	1	0.0	1	33	0.01	<.5	9	<.005	<.5	<.005	1	<.005	<.5	<.005	<.5	<.005	<.5	
108.29×10 <sup>9</sup>		3.1	10	0.05	<.5	0.2	10	0.4	3	63	0.01	<.5	9	0.01	<.5	0.6	10	0.1	1	0.01	<.5	<.005	<.5	
151.46×10 <sup>15</sup>		1503.9	15	0.17	<.5	1.8	15	2.0	10	93	0.37	1	9	0.02	<.5	4.7	15	0.2	2	0.01	<.5	<.005	<.5	
1003.03×10 <sup>105</sup>		—	—	39.70	16	—	—	—	—	603	—	—	9	1.60	62	—	—	—	—	0.62	62	0.62	62	
Round robin: $K = N + 1$ , $ \mathcal{E}  = 6N$																								
53.60×10 <sup>2</sup>		0.1	1	<.005	<.5	0.0	1	0.2	1	19	<.005	<.5	6	<.005	<.5	0.0	1	0.1	1	<.005	<.5	<.005	<.5	
102.30×10 <sup>5</sup>		68.2	11	0.01	<.5	0.2	11	85.0	11	39	0.01	<.5	11	0.01	<.5	0.3	11	78.5	13	<.005	<.5	<.005	<.5	
151.10×10 <sup>6</sup>		4201.5	40	0.02	<.5	0.6	40	4922.7	40	59	0.03	<.5	16	0.01	<.5	1.2	40	4739.5	44	0.01	<.5	<.005	<.5	
1002.85×10 <sup>32</sup>		—	—	1.98	1	—	—	—	—	399	13.32	32	101	4.67	19	—	—	—	—	1.29	20	1.29	20	
Flexible manuf. sys.: $K = 19$ , $ \mathcal{E}  = 20$																								
23.44×10 <sup>3</sup>		128.3	17	0.01	<.5	0.2	17	318.1	43	31	0.04	<.5	6	0.01	<.5	0.2	17	128.9	18	<.005	<.5	<.005	<.5	
34.86×10 <sup>4</sup>		4107.5	127	0.02	<.5	1.0	127	—	—	46	0.16	<.5	8	0.02	<.5	1.0	127	—	—	0.01	<.5	<.005	<.5	
258.54×10 <sup>13</sup>		—	—	17.98	<.5	—	—	—	—	376	—	—	—	52	1010.85	293	—	—	—	50.38	251	50.38	251	

Table 1: Experimental results: SMART vs. NuSMV.

## 5 Conclusion and Future Work

We showed how, by exploiting the structure of a discrete-state model, one can recognize event locality, encode it using a boolean Kronecker matrix representation of the next-state function, and greatly improve the efficiency of symbolic CTL model checking, i.e., the computation of the sets of states satisfying an  $EX$ ,  $EF$ ,  $EU$ , or  $EG$  formula.

Furthermore, we showed that the *saturation* algorithm we initially proposed for state-space generation can be adapted to efficiently find the states satisfying an  $EU$  expression, by automatically classifying the *safety* of the model events. The resulting *EUsat* algorithm is at least as fast, and in many cases much faster than, our already improved  $EU$  computation. *EUsat* can also be used as the key procedure to compute the set of states satisfying an  $EG$  formula. However, this approach enumerates the SCCs in the model, thus it can be pathologically poor.

In the future, we will investigate how to further improve the  $EG$  computation, for example by exploring how saturation may be used to obtain the SCC hull without enumeration. We will also extend our work to Fair-CTL, since our event classification idea should remain applicable. Finally, since the asynchronous systems we target are often studied using explicit partial-order reduction techniques, we intend to perform a thorough comparison with tools such as SPIN.

## References

- [1] R. Alur et al. Partial-order reduction in symbolic state space exploration. In *Proc. CAV*, pages 340–351. Springer, 1997.
- [2] R. Bloem, H. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in  $n \log n$  symbolic steps. In *Proc. FMCAD*, pages 37–54. Springer, 2000.
- [3] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proc. DAC*, pages 29–34. ACM Press, 2000.
- [4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.
- [5] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. Comp.*, 12(3):203–222, 2000.
- [6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. In *LICS*, pages 428–439, 4–7 1990.
- [7] G. Ciardo et al. SMART: Stochastic Model checking Analyzer for Reliability and Timing, User Manual. Available at <http://www.cs.wm.edu/~ciardo/SMART/>.
- [8] G. Ciardo, G. Luetzgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *Proc. ICATPN*, LNCS 1825, pages 103–122. Springer, June 2000.
- [9] G. Ciardo, G. Luetzgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, LNCS 2031, pages 328–342. Springer, Apr. 2001.
- [10] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Proc. TACAS*, LNCS 2619, pages 379–393. Springer, Apr. 2003.



- [11] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. CAV*, LNCS 1633, pages 495–499. Springer, 1999.
- [12] 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*, LNCS 131, pages 52–71. Springer, 1981.
- [13] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
- [14] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Proc. Int. Conference on VLSI*, pages 49–58. IFIP Transactions, North-Holland, Aug. 1991.
- [15] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.
- [16] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Proc. ICCD*, pages 220–223. IEEE Comp. Soc. Press, Sept. 1990.
- [17] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. ICATPN*, LNCS 1639, pages 6–25, June 1999.
- [18] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. SIGMETRICS*, pages 147–153, May 1985.
- [19] K. Ravi and F. Somenzi. Efficient fixpoint computation for invariant checking. In *Proc. ICCD*, pages 467–474. IEEE Comp. Soc. Press, Oct. 1999.
- [20] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proc. ICCAD*, pages 37–40. ACM Press, 1999.

# A Work-Efficient Distributed Algorithm for Reachability Analysis

Orna Grumberg, Tamir Heyman, and Assaf Schuster

Computer Science Department, Technion, Haifa, Israel

**Abstract.** This work presents a novel distributed, symbolic algorithm for reachability analysis that can effectively exploit, “as needed”, a large number of machines working in parallel. The novelty of the algorithm is in its dynamic allocation and reallocation of processes to tasks and in its mechanism for recovery, from local state explosion. As a result, the algorithm is *work-efficient*: it utilizes only those resources that are actually needed. In addition, its high adaptability makes it suitable for exploiting the resources of very large and heterogeneous distributed, non-dedicated environments. Thus, it has the potential of verifying very large systems.

We implemented our algorithm in a tool called Division. Our preliminary experimental results show that the algorithm is indeed work-efficient. Although that the goal of this research is to check larger models, the results also indicate the potential to obtain high speedups, because communication overhead is very small.

## 1 Introduction

Reachability analysis is a central component of model checking. The verification of most temporal safety properties can be reduced to reachability analysis [3]. It is also an important preliminary stage for increasing the efficiency of symbolic model checking.

A significant amount of work is invested in increasing the capacity of model checking. Current model checking tools can verify systems with hundreds of variables using BDD-based methods [6,14] and falsify systems with thousands of variables using SAT-based methods [4]. A recent comparison [1] shows that each of the BDD-based and the SAT-based methods is superior to the other for certain types of problems. Nevertheless, it is generally agreed that the capability of model checking tools should be extended.

Typically, BDD-based model checking tools suffer from high space requirements while SAT-based tools suffer from high time requirements. The goal of this work is to overcome the space problem of BDD-based model checkers. A promising approach is to exploit the accumulative computation power and memory of a number of machines that work in parallel. Many environments can provide a large number of machines whose collective memory exceeds the memory size of any single machine.

Several solutions employing parallel computation have been suggested for dealing with the large memory requirements. Several papers suggest replacing the BDD with a parallelized data structure [19,15]. In [18], an explicit model checker that does not use symbolic methods is parallelized. Other papers suggest reducing the space requirements by partitioning the work to several tasks [8,17,16,7]. Although these methods might, in principle, be parallelized, they have not been. Rather, they use a single computer

to sequentially handle one task at a time, while the other tasks are kept in an external memory.

The work that most resembles this one is distributed symbolic (BDD-based) reachability analysis, suggested in [13]. It is based on an initial partitioning of the state space among all processes in the network and on a continuous load balancing that keeps the workload among the processes relatively balanced.

The success of this approach strongly depends on an effective slicing procedure. Slicing is said to be *effective* if it avoids duplication and if it results in evenly split, smaller BDDs. Duplication is the amount of sharing in a BDD structure that is lost due to partitioning. The notion of duplication and its implications are discussed in detail in [13] and will not be addressed in this paper. Finding such an effective slicing is a nontrivial problem [8,17,16,13].

In [13], each process iteratively applies *image computation* to its set of *new states*  $N$ , *exchanging* non-owned states with other processes, and collecting owned states in its set of *reachable states*  $R$ . Load balance is available at the end of each iteration. It balances the sizes of the sets of reachable states in the different processes.

This algorithm has several drawbacks. First, it immediately splits to as many slices as the number of processes in the network and does not release them until it terminates. Thus, it occupies all processes in the network all the time, regardless of actual need. Second, slicing is often inefficient because it partitions a relatively small BDD into many small slices. The more processes in the system, the less efficient the slicing is, which renders the algorithm non-scalable. Third, it does not provide a means to overcome the memory overflow that occurs during an image computation or an exchange operation. It is well known that intermediate results of image computation may be orders of magnitude larger than its initial and resulting BDDs. Similarly, during an exchange operation the memory of a process may overflow as a result of the BDDs it receives. Unfortunately, even when there are under-utilized processes, there is no way to recover from such overflows since load balancing is available only at the end of iterations. Finally, balancing is applied only to the sets  $R$ . However, the size of intermediate results in image computation depends on  $N$  and is often much larger than  $R$ . Thus, load balancing does not handle the dominant factors of memory overflow.

In this paper we suggest a new algorithm which overcomes the drawbacks of the previous one. The algorithm uses two types of processes: coordinators and workers. Each worker can be either active or free. The algorithm works iteratively. It is initialized with one active worker that runs a symbolic reachability algorithm, starting from the set of initial states. During its run, workers are allocated and freed, as needed. At any iteration, each of the active workers applies image computation and then sends those states it does not own to their owners. Therefore, we will refer to these as a worker's non-owned states.

Since memory overflow is likely to occur during the image computation and the exchange operation, our algorithm is designed to overcome these problems. For image computation we use a new BDD operation that resembles ordinary image computation, except that it stops if the intermediate results create memory overflow. In this case, the BDD representing the intermediate results is partitioned into  $k$  slices. One slice is left with the overflowed worker and the others are distributed to  $k - 1$  free colleagues.  $k$  is

called the *splitting degree*. It is a parameter of the new algorithm and is usually small (often  $k = 2$ ). Since the BDD is huge, the slicing is very effective. Once the BDD is split, each worker resumes the computation of (its part of) the image *from the point at which it stopped*. However, each worker now works on a smaller BDD. If state explosion occurs during the exchange procedure, then  $R \cup N$  is split for sharing with  $k - 1$  free colleagues. Exchanging of non-owned states then proceeds according to the new ownership.

The new algorithm enables the slicing procedure to split according to  $R$ ,  $N$ , or intermediate results, depending on what caused the memory overflow. Since the chosen BDDs are large, slicing is always very effective. Furthermore, slicing affects the performance of the new algorithm much less than it affects the one from [13] because, in the case of a high work load at one of the co-workers, the new algorithm can simply split again. These features provide the new algorithm with strength and flexibility, and allow to reduce the slicing complexity.

It may also happen that the memory requirement of a worker decreased below a certain threshold (the size of a BDD decrease even if it represents a larger set of states). In that case, several workers with small memory requirements are combined and all but one become free.

It is important to note that splitting occurs only “as needed”, when a worker actually has a memory overflow. Thus the algorithm is *work-efficient*: it exploits to the maximum the resources of the active workers before allocating additional ones. This efficiency allows, for a given network, computing reachability of (i.e., verifying) larger systems. Moreover, our algorithm can effectively exploit any network size. Thus, the larger the available network, the larger the systems that can be verified.

We have implemented our algorithm in Division, a generic platform for the study of distributed symbolic model checking [12]. Division requires a model checker as an external module. We used NuSMV [10] for this purpose: a re-implementation of McMillan’s SMV [14].

Unfortunately, using NuSMV implied that we could not directly compare the results of [13] to the results of this work. The experiments in [13] were conducted using the high-performance RuleBase [2] model checker that was not available to us in this work. The two tools are not comparable as many of the RuleBase optimizations are not implemented in NuSMV.

Our parallel testbed included 25 dual process PC machines. The nodes communicated via a fast Ethernet connection. We conducted our experiments using four of the largest circuits from the ISCAS89 and addendum’93 benchmarks.

With our distributed algorithm, we can compute larger models than we can compute with a single machine using the same model checker. In all the examples the new algorithm using the less sophisticated model checker (NuSMV) would be sufficient to compute the same models and reach the same BFS step as in [13].

The rest of the paper is organized as follows. In Section 2 we detail the algorithm that the workers follow. Section 3 describes the operation of the coordinator processes. Section 4 explains the enhanced slicing employed when overflow occurs during image computation. Preliminary experimental results are given in Section 5. We summarize our conclusions and expectations in Section 6.

## 2 The Worker Algorithm

Our distributed algorithm uses a set of window functions [8,17] to partition the state space among all workers in the network. Each worker *owns* the states in one of the window functions and computes the reachable states in this window.

Figure 1 presents a high-level view of the workers algorithm. Essentially, the algorithm performs a reachability task. The algorithm starts with only one worker that owns the entire state space, while the rest of the workers are free. If a worker runs out of memory (*memory overflow*), it distributes parts of its work among a few free workers.

The worker repeatedly computes images and sends its non-owned states to their owners until termination is detected (namely, a fixed-point is reached). While iterating, if the workload of the worker becomes too small, it participates in a `collect_small` procedure.

There are two points at which a worker may run out of memory (*memory overflow*): during the image computation and during the exchange of non-owned states. Upon memory overflow, the worker splits the states it owns into two parts: one that will be processed at the current worker and another to be processed at another worker. As a result, the states belonging to the new worker become non-owned and are sent out to the new worker.

```
function reach_task()
1 Loop until termination()
2 Image() if overflow, split and use new workers
3 Exchange() if overflow, split and use new workers
4 Collect_small()
5 return owned states
```

**Fig. 1.** High-level pseudo-code for a worker

Let us describe the algorithm for the workers in greater detail, as shown in Figure 2. The reachability task includes a set of reachable states  $R$  and a set of reachable states that are not yet developed,  $N$ . For brevity, we omit in this section the worker subscript  $id$  from  $R_{id}$  and  $N_{id}$ , as well as the window function  $w_{id}$ . The set  $R$  is included in a window function  $w$ . The sets  $R$  and  $N$ , as well as the window function  $w$ , may change during the algorithm's execution.

In the `Image` procedure, the worker computes the set of states that can be reached in one step from  $N$  and stores the result in a new  $N$ . However, if during image computation the memory overflows, the worker splits  $w$  and updates  $R$  and  $N$  accordingly, as described below.

In the `Exchange` procedure the worker uses  $w$  to define the part of the state space it “owns”. It sends out the non-owned states ( $N \setminus w$ ) to their owners and receives its owned states that were found by other workers.

Finally, if only a small amount of work remains, the worker joins the `Collect_small` procedure. The `collect_small` procedure adds up the tasks of several workers, each of which has only a small amount of work. This is done by joining together the parts of the state space owned by those workers and assigning the unified ownership to one of them. The others become “free” ( $w = \emptyset$ ) and return to the pool of free workers.

```

function reach_task( $R, w, N, \text{method}$ )
  if  $\text{method} = \text{"exchange"}$ 
    goto Exchange_loop( $R, w, N$ )
  Loop_forever
    Image( $R, w, N$ )
    Exchange( $R, w, N$ )
    if (termination()) return  $R$ 
     $N = N \setminus R$ 
     $R = R \cup N$ 
    Collect_small( $R, w, N$ )
    if ( $w = \emptyset$ )
      send  $\langle \text{to\_pool}, \text{id} \rangle$  to  $\text{ex\_coor}$ 
      return to pool

procedure Exchange( $R, w, N$ )
   $\langle \{w_i\} \rangle = \text{receive from ex\_coor}$ 
  send  $\langle \{p_i\} \rangle$  to  $\text{ex\_coor}$ 
  Exchange_loop( $R, w, N$ )

procedure Collect_Small( $R, w, N$ )
  While( $(|N| + |R| < \text{Min})$ )
    send  $\langle (|N|, |R|) \rangle$  to  $\text{small\_coor}$ 
     $\langle \text{action} \rangle = \text{receive from small\_coor}$ 
    if  $\text{action} = \langle \text{End} \rangle$  return
    if  $\text{action} = \langle \text{Non\_owner}, p_{clg} \rangle$ 
      send  $\langle R, w, N \rangle$  to  $p_{clg}$ 
       $R = w = N = \emptyset$ 
       $\langle \text{"release"} \rangle = \text{receive from ex\_coor}$ 
      return
    if  $\text{action} = \langle \text{Owner}, p_{clg} \rangle$ 
       $\langle R', w', N' \rangle = \text{receive from } p_{clg}$ 
       $R = R \cup R'; w = w \cup w'; N = N \cup N'$ 
      send  $\langle w, p_{id}, p_{clg} \rangle$  to  $\text{ex\_coor}$ 

procedure Image( $R, w, N$ )
   $N = \text{boundedImage}(N, \text{Max}, \text{Failed})$ 
  While( $\text{Failed}$ )
    Split( $R, w, N, \text{"Image"}$ )
     $N = \text{boundedImage}(N, \text{Max}, \text{Failed})$ 

procedure Exchange_Loop( $R, w, N$ )
  loop until  $\langle \text{done} \rangle$  received from  $\text{ex\_coor}$ 
     $\langle p_{clg}, w_{clg} \rangle = \text{receive from ex\_coor}$ 
    send  $\langle N \cap w_{clg} \rangle$  to  $p_{clg}$ 
     $\langle N' \rangle = \text{receive from } p_{clg}$ 
     $\text{overflow} = N'$  is too large
    send  $\langle \text{overflow} \rangle$  to  $p_{clg}$ 
    send  $\langle \text{status} \rangle = \text{receive from } p_{clg}$  to  $\text{ex\_coor}$ 
    if ( $\text{overflow}$ ) Split( $R, w, N, \text{"Exchange"}$ )
    else
       $N = N \cup N'$ 
      send  $\langle \text{"done"} \rangle$  to  $\text{ex\_coor}$ 

procedure Split( $R, w, N, \text{method}$ )
   $\langle \{p_2 \dots p_k\} \rangle = \text{receive from pool\_mgr}$ 
  if ( $\text{method} = \text{"exchange"}$ )
     $\{W'_i\} = \{N w'_i\} = \text{Slice}(R \cup N, k)$ 
  else
    if ( $|R|$  big enough)
       $\{W'_i\} = \text{Slice}(R, k)$ 
    else
       $\{W'_i\} = \emptyset, i \in 2 \dots k; W'_1 = w$ 
       $\{N w'_i\} = \text{Slice}(N, k)$ 
   $\forall i \in 2 \dots k$ :
    send  $\langle R \cap W'_i, w \cap W'_i, N \cap N w'_i, \text{method} \rangle$  to  $p_i$ 
     $R = R \cap W'_1; w = w \cap W'_1; N = N \cap N w'_1$ 
    send  $\langle \{i \in 1..k \mid w \cap W'_i\} \rangle$  to  $\text{ex\_coor}$ 

```

**Fig. 2.** Pseudo-code for a worker in the distributed reachability computation

In the Image procedure, the image is computed using a new BDD operation. The Image procedure is using a new BDD operation,  $\text{boundedImage}(N, \text{Max}, \text{Failed})$ . This operation is different from traditional image computation in that it stops the local computation in case of a memory overflow (i.e., the number of BDD nodes exceeds  $\text{Max}$ ). Upon overflow, the Image procedure calls the Split procedure, which repartitions the ownership of the worker and updates  $R, w, N$  accordingly.

In the Exchange procedure, the worker first requests and receives from the  $\text{ex\_coor}$  process the up-to-date list of window functions owned by the other workers. The worker then sends the  $\text{ex\_coor}$  the list of workers to whom it wishes to send non-owned states. Then, in the Exchange\_loop procedure, the  $\text{ex\_coor}$  schedules the worker for state exchange with other workers.

In the Exchange\_loop procedure the worker is scheduled by the  $\text{ex\_coor}$  to exchange non-owned states with colleagues that either found states owned by the worker or own states that were found by the worker. The worker continues to receive exchange commands from the  $\text{ex\_coor}$  until it gets a  $\langle \text{done} \rangle$  command when there are no more pending exchanges. If the worker's memory overflows during the exchange procedure and the worker fails to receive more owned states, it notifies the  $\text{ex\_coor}$  and calls the Split procedure to reduce its ownership.

If the worker in the `Collect_Small` procedure has enough work, it exits immediately. Otherwise, the worker notifies the `small_coord` about the sizes of its  $N$  and  $R$  sets. In reply, it receives one of three commands and proceeds accordingly:  $\langle End \rangle$  commands it to exit the `Collect_Small`;  $\langle Non\_owner, p_{clg} \rangle$  commands it to deliver its ownership and owned states to a colleague worker  $p_{clg}$ , waits for the `ex_coord` to acknowledge the update of its window functions (performed by  $p_{clg}$ ), and then return to the pool;  $\langle Owner, p_{clg} \rangle$  commands it to take over the ownership and states of another worker  $p_{clg}$  and report the new ownership to the `ex_coord`.

The `Split` procedure starts by requesting from the `pool_mgr`  $k - 1$  new workers (which, together with the overflowed worker, makes it a  $k$ -way split). If `Split` is called from `Exchange`, then the window function  $w$  of the overflowed worker is split into  $k$  new window functions  $\{W'_i\}$ , such that  $\{W'_i \cap R\}$  have approximately the same sizes. If `Split` is called from `Image`, then two sets of  $k$  new window functions are computed, as follows. If  $R$  is big enough, then, as in the previous case, a set of window functions  $\{W'_i\}$  is computed such that the sizes of  $\{W'_i \cap R\}$  are approximately the same. Otherwise, if  $R$  is too small, one of the workers gets all of  $w$  while the others remain empty. In any case, the  $i$ th new window function  $W'_i$  determines, for the  $i$ th worker, its new window  $w_i$ . In addition,  $w$  is split again into another set of window functions  $\{Nw'_i\}$ , this time making  $\{Nw'_i \cap N\}$  equal in size. After the new window functions are computed, the overflowed worker sends the corresponding states to its new colleagues.

The reason for computing two different partitions when `Image` overflows is that  $\{Nw'_i\}$  attempts to balance the current image computation, while  $\{W'_i\}$  attempts to balance the memory requirement in the full reachability process. In section 4 we further discuss the optimization of the partitioning process.

In the case that  $R$  is "too small" or even empty, the new colleagues are simply helping the overflowed worker with a single image computation. Once the image is computed, all states produced by the helpers are non-owned and will be sent to other workers that own them. From our experience, this case is not uncommon; it occurs when the peak memory requirement during image computation is much larger than  $R$ .

As mentioned in the introduction, an important advantage of our algorithm over previous works is that it calls the `Slice` function only when the memory overflows, and with  $k$  much smaller than the total number of workers. This makes slicing much more effective in producing even splits of the input sets of states.

We remark that the `Slice` procedure itself is no different from the slicing mechanisms described in [13]. Thus, in this paper, we use it as a black box and focus on the distributed algorithm itself.

### 3 The Coordinators

The `ex_coord` coordinator holds the current set of window functions and coordinates the exchange of non-owned states between workers. In order to hold a consistent view of the current set of window functions, the `ex_coord` is notified immediately on every split or merge of windows. It takes the following actions on incoming event notifications:

- When a worker requests an exchange it first *registers* at the `ex_coor`. The `ex_coor` replies with the up-to-date set of window functions and receives in return the set of colleagues the worker wants to communicate with.
- When a worker splits, the `ex_coor` updates the set of window functions. If the splitting worker is already registered for exchange states, the `ex_coor` notifies all the workers that have asked to send it states that they should send the states to the new set of workers, according to the new set of window functions.
- When workers perform *Collect.Small* and join their ownerships, the `ex_coor` updates the set of window functions. If there are workers registered for exchanging states with the joining workers, the `ex_coor` redirects them to the new owner. When the `ex_coor` complete to update the set of window functions it sends `< release >` command to the worker that become non-owner.
- When a worker completes the exchange of non-owned states with another worker, the coordinator marks it as available for another round of exchange states.
- When a worker asks to re-launch an exchange because the colleague overflowed and had to split while they were interacting, the `ex_coor` adds this request to the list of exchange requests.

The `small_coor` coordinator collaborates with `ex_coor` to prevent deadlocks and to collect as many under-utilized workers as possible. The `small_coor` receives registration requests from workers that completed the exchange phase and are left with a very low load (very small  $R \cup N$ ). The first registrant is blocked until more of them arrive. When there are several registrants the `small_coor` instructs them to merge.

The `pool_mgr` coordinator keeps track of free workers. During initialization, the `pool_mgr` marks all but one worker as free. When a worker invokes the Split procedure, it sends a request to the `pool_mgr` for  $k - 1$  free workers (where  $k$  is the splitting degree). The `pool_mgr` replies with a list of  $k - 1$  worker *ids* and removes them from the free list. Throughout the algorithm, when a worker becomes free, i.e., when its ownership becomes empty, it returns to the `pool_mgr` and is added to the free list for later assignments.

If at the time free workers are requested from the `pool_mgr`, the free list happens to be empty or is shorter than  $k - 1$ , the `pool_mgr` announces a “worker overflow” and stops the execution globally.

## 4 Optimizing the Splitting in Image Computation Overflow

Our algorithm is based on the assumption that in case of a memory overflow during image computation, splitting the window of the overflowing worker enables the completion of the computation using more workers. The current splitting method strives to effectively slice the set  $N$  on which the image is computed (see [13]). However, since the computation is symbolic, reducing the size of the subsets does not guarantee a corresponding reduction in the image size. Furthermore, it guarantees even less for the size of the intermediate results that commonly dictate the *peak memory requirement* during the image computation. Our experience shows that even when the size of the parts is the same, the size of the peaks may differ greatly. Thus, while one of the slices may have no problem in completing the image computation, another may overflow again.



Another problem with the current splitting method is the time penalty for memory overflow. When the image computation overflows and the set  $N$  is split, the work that was invested in the current image step is lost, and the work is repeated all over again. In fact, in the case of several subsequent memory overflows, the work is repeated again and again. Notice that the ratio between the peak memory requirement in the image computation and the set  $N$  is commonly two or three orders of magnitude. Thus, memory overflow commonly occurs when a big part of the image computation has already been done locally, and all this work must be repeated. Since the image computation takes most of the time in our distributed algorithm, the repeated work slows down the algorithm substantially.

The solution to the above two problems is simply to split the intermediate results and not the set  $N$ . After the splitting, the parts of the intermediate results are distributed among the new workers, so computing the image for each of them continues from the point of the overflow. In this way there is no time penalty for overflow except for the splitting computation (which is of somewhat higher complexity than before). Of course, communicating the intermediate results requires a much higher bandwidth. However, network bandwidth and communication delay turn out to be minor factors as compared with the time spent in the image computation, even with our standard fast Ethernet.

In terms of memory requirements this solution has two advantages. First, splitting is applied on a much larger set, which makes it a lot easier to split effectively. Second, splitting is applied much closer to the peak, which makes it more efficient in reducing the peak memory requirements of the resulting parts.

The optimized algorithm uses a partitioned transition relation. The full transition relation is a conjunction of all partitions:

$$T(V, V') = T_1(V, V') \wedge T_2(V, V') \wedge \dots \wedge T_n(V, V'),$$

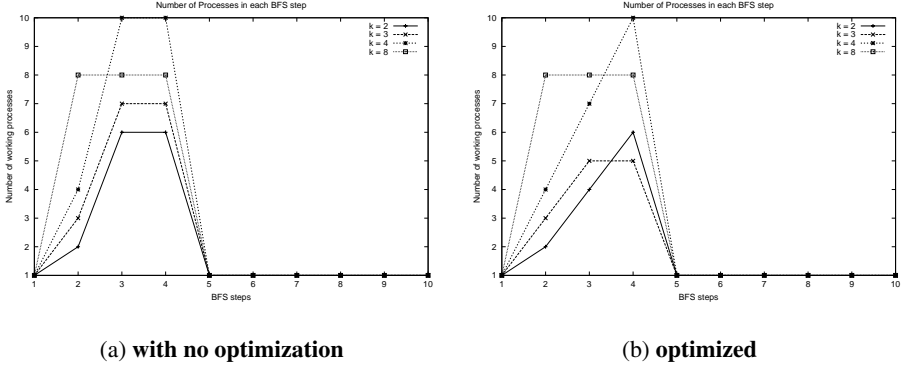
and an image computation thus becomes

$$S'(V') = \exists V[S(V) \wedge T_1(V, V') \wedge T_2(V, V') \wedge \dots \wedge T_n(V, V')].$$

The technique for image computation suggested by Burch et al. [5] is to iteratively conjunct-in the partitions, and to quantify-out variables as soon as further steps do not depend on them. The order in which  $T_i(V, V')$  are conjuncted is very important to the efficiency of this technique [11]. For the sake of simplicity, let us assume the order is given such that  $T_1$  is the first to conjunct, then  $T_2$ , until  $T_n$ . Let  $D_i$  be the set of variables on which  $T_i(V, V')$  depend. Let  $E_i = D_i - \bigcup_{m=i+1}^n D_m$ . A symbolic step is carried out iteratively as follows:

$$\begin{aligned} S_1(V, V') &= \exists E_1[T_1(V, V') \wedge S(V)] \\ S_2(V, V') &= \exists E_2[T_2(V, V') \wedge S_1(V, V')] \\ &\vdots \\ S'(V') &= \exists E_n[T_n(V, V') \wedge S_{n-1}(V, V')]. \end{aligned}$$

If overflow occurs during step  $0 < j < n$ , we look for a set of window functions  $w_1 \dots w_k$  such that  $\bigvee_{i=1}^k w_i = 1$ . The  $i$ th worker will get  $S_j(V, V') \wedge w_i$ . We can now



**Fig. 3.** Number of workers required in each BFS step of s1269. Overflow is declared for worker memory utilization exceeding 6M BDD nodes.

rewrite the  $j + 1$  step as follows:

$$S_{j+1}(V, V') = \exists E_{j+1} \left[ \bigvee_{i=1}^k T_{j+1}(V, V') \wedge S_j(V, V') \wedge w_i \right].$$

Since the existential quantification is distributive over disjunction, the above expression is equal to:

$$S_{j+1}(V, V') = \bigvee_{i=1}^k \exists E_{j+1} [T_{j+1}(V, V') \wedge S_j(V, V') \wedge w_i].$$

Therefore, the disjunction of the  $j + 1$ th steps assigned to each worker is equal to the step done without splitting.

The algorithm uses a new BDD operation:  $\text{BoundInc}(S(V, V'), \{T_i(V, V')\}, \text{Max})$ , where  $S(V, V')$  is the function from which the image computation continues,  $\{T_i(V, V')\}$  is the set of partitions that were not yet used, and  $\text{Max}$  is the threshold for overflow during image computation. In the beginning of the algorithm,  $S(V, V')$  is the set of states whose image is to be computed in this step, and  $\{T_i(V, V')\}$  are all the partitions. If the algorithm overflows,  $\text{BoundInc}$  returns in  $S(V, V')$  the last intermediate result computed prior to the overflow, and in  $\{T_i(V, V')\}$  the rest of the partitions that have not been used. If the algorithm completes the image computation,  $S(V, V')$  equals the next set of states, and an empty list of partitions is returned.

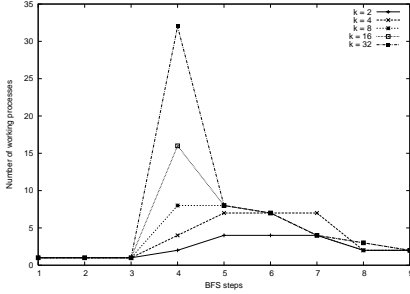
Figure 3 illustrates the benefit of using the optimized algorithm for the circuit s1269. Figure 3(a) provides the number of workers required in each step for various splitting degrees. For instance, for splitting degree  $k = 2$ , six workers are needed in order to complete Step 3. Figure 3(b) shows that this step requires only four workers when using the optimization described in this section. In all other steps and splitting degrees the number of workers required by the optimized algorithm was always less than or equal to the non-optimized version.

## 5 Experimental Results

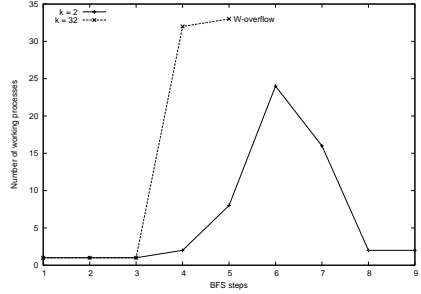
Our parallel testbed included 25 PC machines, each consisting of dual 1.7GHz Pentium 4 processors with 1GB memory. The communication between the nodes consisted of a fast Ethernet. We conducted our experiments using four of the largest circuits from the ISCAS89 benchmarks. The characteristics of the circuits are given in Figure 4.

Circuit	#vars	peak		fixed point	
		size	step	time	steps
prolog	117	2.6M	5	2,431	9
s1269	55	16M	5	5,053	10
s3330	172	16M >	Ov(3)	-	Ov(3)
s1423	88	16M >	Ov(13)	-	Ov(13)

**Fig. 4.** Benchmark suite characteristics. The peak is the maximal memory requirement at any point during an image step. Fixed point is the number of image steps and the time (seconds) it takes to get to the fixed point. Ov( $m$ ) denotes memory overflow at step  $m$ .



(a) **prolog**  $Max = 1M$  nodes allocated



(b) **S3330**  $Max = 7M$  nodes allocated

**Fig. 5.** Number of workers in each BFS step. Overflow is declared for worker memory utilization exceeding  $Max$  BDD nodes. **W-overflow** halts the computation when more than 60 workers are required.

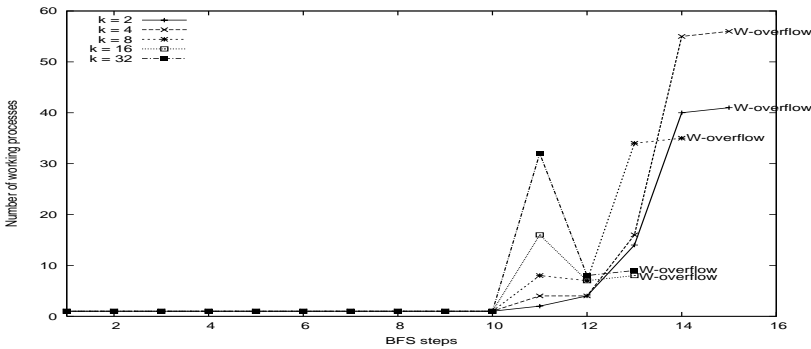
### 5.1 Number of Workers for Reachability Analysis

Since the memory required by each worker is bounded by a given threshold, we only care about the number of active workers at each iteration. Figures 5(a), 5(b), 6 and 3 give the number of workers required at any step of the analysis, and the threshold that was used. The figures prove that using a lower splitting degree is more work efficient, namely, the computation can be carried using fewer resources with a lower splitting

degree. This is explained by the fact that when the splitting degree is high, new workers may join in even when the computation can do without them: the computation proceeds with workers that may be under-utilized (but not sufficiently so to be collected by the `Collect_Small` process).

In steps 1, 2, 3 in Figure 5(a) only one worker is needed. In step 4, this worker needs help in order to complete the image computation. Dividing the work into two is sufficient, but when the splitting degree is higher we occupy more workers without actually needing them. In steps 8 and 9 the image computation requires less memory and the size of the sets  $R$  and  $N$  requires less workers. Indeed the number of workers decreases as a result of the `Collect_Small` procedure.

Figure 5(b) shows that the distributed system can complete the reachability analysis, whereas a single machine overflows.



**Fig. 6.** Number of workers in each BFS step of `s1423`. Overflow is declared for worker memory utilization exceeding 6M BDD nodes. **W-overflow** is where more than 60 workers required.

## 5.2 Timing and Communication

We have performed some initial studies regarding the timing and breakdown of running our distributed system. The results show several very clear findings and trends that we now briefly discuss.

First, communication overhead is minor. Our experiments show that the time to reach local overflow is much higher than the time required to dump the contents of memory into the network. Although this finding should be re-evaluated when our system is further optimized (see below), it seems strong enough to sustain. If the system scales up to include more workers, the communication time might grow as a result of more non-owned states that are found. Nevertheless, we expect the computation time to remain dominant because the communication volume for every worker at any split or exchange operation is bounded by the size of the RAM of that worker. We remark that technology trends predict much faster commodity networks (even when compared to the larger expected RAMs) very soon.

Second, splitting is a major element in the computation. It can count up to dozens of percentage points of the computation time, and these numbers grow rapidly when the system scales up. Others have previously addressed the splitting complexity [9]; we intend to speed up the splitting module in our future work.

Third, the fact that the reachability computation is synchronized in a step-by-step fashion has a major impact on the computation time. The problem is that at the end of a step all computing workers wait for the slowest one, who may be slicing and re-slicing several times during the step (remember that slicing is slow!). However, despite its synchronized operation, the new algorithm is very flexible. We believe that it can become the basis for a truly non-synchronized variant.

One interesting phenomena that was not masked by the inefficiencies above is a tradeoff between being work efficient and obtaining speedups. While the best hardware utilization is achieved with splitting degree of 2, the fastest computation times are obtained using somewhat higher splitting degrees (e.g.,  $k = 8$  for Prolog). Thus, a splitting degree higher than 2 may become instrumental in cases that the speedup is more important than RAM utilization.

## 6 Conclusions and Expectations

This paper presents a new distributed algorithm for symbolic reachability analysis that improves significantly on previous works. Its adaptability to any network size and its high utilization of network resources make it suitable for solving very large verification problems.

The experimental environment that is used to evaluate our new algorithm currently consists of NuSMV and the newly introduced Division system. Division is a new platform for distributed symbolic model checking research, featuring a high-level generic interface to “external” model checkers. Eventually, we intend to release Division source code through the Division web-site [12].

At the point that the final version of this paper is due, Division is in the final stages of interfacing with Intel’s high-performance model checker – Forest. We thus expect our results to improve substantially and to become more accurate in the near future. We refer the interested reader to the Division web-site for up-to-date result reports and for the full and final version of this paper.

## References

1. N. Amla, R. Kurshan, K. McMillan, and Medel R. K. Experimental Analysis of Different Techniques for Bounded Model Checking. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’03)*, LNCS, Warsaw, Poland, 2003.
2. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An Industry-Oriented Formal Verification Tool. In *33rd Design Automation Conf.*, pages 655–660, 1996.
3. I. Beer, S. Ben-David, and A. Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, LNCS 818, 1998.
4. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, 5<sup>th</sup> Int. Conference, TACAS’99*, LNCS 1579, 1999.

5. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proc. of the 1991 Int. Conference on Very Large Scale Integration*, August 1991.
6. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking:  $10^{20}$  states and beyond. *Information and Computation*, 98(2):142–171, June 1992.
7. G. Cabodi. Meta-BDDs: A Decomposed Representation for Layered Symbolic Manipulation of Boolean Functions. In *Proc. of the 13th Int. Conf. on Computer Aided Verification*, 2001.
8. G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large FSM. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
9. G. Cabodi, P. Camurati, and S. Quer. Improving the Efficiency of BDD-Based Operators by Means of Partitioning. *IEEE Transactions on Computer-Aided Design*, May 1999.
10. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proc. of the 7th Int. Conf. on Computer-Aided Verification (CAV'99)*, LNCS 1633, pages 495–499, Trento, Italy, 1999.
11. D. Geist and I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Proc. of the Sixth Int. Conf. on Computer Aided Verification*, LNCS 818, pages 299–310, 1994.
12. O. Grumberg, A. Heyman, T. Heyman, and A. Schuster. Division System: A General Platform for Distributed Symbolic Model Checking Research, 2003.  
[http://www.cs.technion.ac.il/Labs/dsl/projects/division\\_web/division.htm](http://www.cs.technion.ac.il/Labs/dsl/projects/division_web/division.htm).
13. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits. *Formal Methods in System Design*, 21(2):317–338, November 2002.
14. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
15. K. Milvang-Jensen and A. J. Hu. BDDNOW: A Parallel BDD Package. In *Second Int. Conference on Formal methods in Computer-Aided Design (FMCAD '98)*, LNCS, Palo Alto, California, USA, November 1998.
16. A. Narayan, A. Isles, J. Jain, R. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
17. A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs. In *Proc. of the IEEE Int. Conf. on Computer Aided Design*, pages 547–554. IEEE Computer Society Press, June 1996.
18. Ulrich Stern and David L. Dill. Parallelizing the Murphy Verifier. In *Proc. of the 9th Int. Conf. on Computer Aided Verification*, LNCS 1254, pages 256–267, 1997.
19. T. Stornetta and F. Brewer. Implementation of an Efficient Parallel BDD Package. In *33rd Design Automation Conf.* IEEE Computer Society Press, 1996.

# Modular Strategies for Infinite Games on Recursive Graphs<sup>\*</sup>

Rajeev Alur<sup>1</sup>, Salvatore La Torre<sup>2</sup>, and P. Madhusudan<sup>1</sup>

<sup>1</sup> University of Pennsylvania

<sup>2</sup> Università degli Studi di Salerno

**Abstract.** In this paper, we focus on solving games in recursive game graphs that can model the control flow of sequential programs with recursive procedure calls. The winning condition is given as an  $\omega$ -regular specification over the observable, and, unlike traditional pushdown games, the strategy is required to be *modular*: resolution of choices within a component should not depend on the context in which the component is invoked, but only on the history within the current invocation of the component. We first consider the case when the specification is given as a deterministic Büchi automaton. We show the problem to be decidable, and present a solution based on two-way alternating tree automata with time complexity that is polynomial in the number of internal nodes, exponential in the specification and exponential in the number of exits of the components. We show that the problem is EXPTIME-complete in general, and NP-complete for fixed-size specifications. Then, we show that the same complexity bounds apply if the specification is given as a *universal* co-Büchi automaton. Finally, for specifications given as formulas of linear temporal logic LTL, we obtain a synthesis algorithm that is doubly-exponential in the formula and singly exponential in the number of exits of components.

## 1 Introduction

An interesting class of infinite-state systems that permits algorithmic verification is *pushdown systems*. Pushdown systems can model the control flow in sequential imperative programming languages with recursive procedure calls. Their analysis has been well studied theoretically [6,20], and forms the basis of recent tools for software verification [4,11,12]. In this paper, we focus on solving *games* over such models. The original motivation for studying games in the context of formal analysis of systems comes from the *controller synthesis* problem [7,16,17]: given a model of the system where some of the choices depend upon the controllable inputs and some of the choices represent uncontrollable nondeterminism, synthesizing a controller that supplies inputs to the system so that the product of the

---

<sup>\*</sup> This research was supported in part by ARO URI award DAAD19-01-1-0473, NSF award CCR99-70925, and NSF award ITR/SY 0121431. The second author was also supported by the MIUR in the framework of project “Metodi Formali per la Sicurezza e il Tempo” (MEFISTO) and MIUR grant 60% 2002.

controller and the system satisfies the correctness specification corresponds to computing winning strategies in two-player games. Besides the long-term dream of synthesizing correct programs from formal specifications, games are relevant for modular verification of open systems. For instance, *Alternating Temporal Logic* allows specification of requirements such as “module A can ensure delivery of the message no matter how module B behaves” [2]; *module checking* deals with the problem of checking whether a module behaves correctly no matter in which environment it is placed [14]; and the framework of interface automata allows assumptions about the usage of a component to be built into the specification of the interface of the component, and formulates compatibility of interfaces using games [10].

Among the many roughly equivalent formulations of pushdown systems, we use the model of *recursive state machines* (RSMs) [1,5]. A recursive state machine consists of a set of component machines called *modules*. Each module has a set of *nodes* (atomic states) and *boxes* (each of which is mapped to a module), a well-defined interface consisting of *entry* and *exit* nodes, and edges connecting nodes/boxes. An edge entering a box models the invocation of the module associated with the box, and an edge leaving a box corresponds to a return from that module. To define two-player games on recursive state machines, the nodes are partitioned into two sets such that a player gets to choose the transition when the current node belongs to its own partition. While the complexity of pushdown games has already been studied [8,20], existing algorithms for solving pushdown games assume that each player has access to the entire global history which includes the information of the play in all modules. In a recent paper, we introduced the notion of *modular* strategies for games on RSMs [3]. A modular strategy is a strategy that has only local memory, and thus, resolution of choices within a module does not depend on the context in which the module is invoked, but only on the history within the current invocation of the module. This permits a natural definition of synthesis of recursive controllers: a controller for a module can be plugged into any context where the module is invoked. Recent work on the interface compatibility checking for software modules implements the global games on pushdown systems [10], but we believe that checking for existence of modular strategies matches better with the intuition for compatibility.

In [3], we showed that solving modular reachability games is NP-complete. In this paper, we consider the general case where the winning condition is specified as an  $\omega$ -regular language over the observable, using Büchi automata or linear temporal logic formulas. Compared to reachability games, there are two additional technical hurdles now. First, the winning strategy needs to produce accepting cycles. Second, the specification is external, and we require the *same* strategy to be used every time a module is invoked. Consequently, if we take the product of the game graph and the specification automaton, as is done typically, we need to find a winning strategy over the product graph that is modular as well as oblivious to the state of the specification.

Our first result is that for a specification given as a deterministic Büchi automaton  $\mathcal{B}$ , the problem of deciding whether there exists a modular strategy



so that the resulting plays are guaranteed to be accepted by  $\mathcal{B}$  is EXPTIME-complete. The upper bound is established using an automata-theoretic solution. We first define strategy trees that encode modular strategies in the recursive game graph  $A$  in a particular way. The strategy tree contains a subtree corresponding to each module. The subtree corresponding to a module  $A_m$  specifies the choice at every existential node of  $A_m$ , and when it enters a box  $b$  corresponding to an invocation of another module  $A_{m'}$ , it just specifies the exit nodes of  $A_{m'}$  that are guaranteed *to be avoided* by the strategy for  $A_{m'}$ . The next step is to define a *two-way alternating tree automaton*  $T$  that accepts the winning strategy trees. An important task of this automaton is to check the consistency of the input tree with respect to the exits to be avoided in each module. The *alternating* nature allows sending of multiple copies, and the two-way nature is exploited to move up and down the tree, rereading strategies of modules as and when needed, and thereby making sure that only one strategy for each module is used. Using the exponential translation from two-way alternating tree automata to nondeterministic tree automata [18], and then, employing the polynomial-time algorithm for checking emptiness of nondeterministic tree automata, we get a complexity bound that is polynomial in the number of internal nodes of  $A$ , exponential in the specification automaton  $\mathcal{B}$ , and exponential in the total number of exit nodes in  $A$ . The exponential dependence on the number of exits seems unavoidable as even reachability games are NP-hard [3]. We show that the NP upper bound applies for fixed-size specifications given by Büchi automata.

Our second result is that given a recursive game graph  $A$  whose nodes are labeled with atomic propositions, and a formula  $\varphi$  of linear temporal logic LTL, the problem of deciding whether there exists a modular strategy in  $A$  so that the resulting paths are guaranteed to satisfy  $\varphi$ , is 2EXPTIME-complete. The hardness holds even for ordinary game graphs [16]. Since translation from LTL to deterministic parity automata is doubly-exponential, using our construction for deterministic specifications would lead to a 3EXPTIME upper bound. We show that the construction for deterministic Büchi specifications can be modified to *universal* Büchi as well as co-Büchi specifications with the same complexity. Unlike a nondeterministic automaton, a universal automaton accepts a word if all runs over that word are accepting. Since the set of models of an LTL formula can be characterized by a universal co-Büchi automaton that is only exponential in the formula, we get a 2EXPTIME bound for LTL games.

We recall that two-way alternating tree automata have been used to solve synthesis problems for pushdown systems with respect to specifications given by  $\mu$ -calculus formulas [13] and parity pushdown games [9]. Apart from the fact that these papers only solve for global strategies and not modular ones, the encodings of strategies are different. While in their encodings, paths to nodes of the tree represent the stack content and the two-way nature is used to push and pop elements, in our setup paths to nodes represent the local history and the two-way nature is used to re-read strategies for a module.

Due to the lack of space we omit some details in this version and refer the interested reader to the full paper.

## 2 Games on Recursive Graphs

Let us fix a finite alphabet  $\Sigma$ . Let  $\Sigma^*$  denote the finite words and  $\Sigma^\omega$  denote the set of  $\omega$ -words over  $\Sigma$ . For any  $n \in \mathbb{N}$ , let  $[n]$  denote the set  $\{1, \dots, n\}$ .

**Definition 1.** A recursive game graph  $A$  (for short *RGG*) over  $\Sigma$  is a tuple  $(M, m_{in}, \{A_m\}_{m \in M})$ , where  $M$  is a finite set of module names,  $m_{in} \in M$  is the name of the initial module and for every  $m \in M$ ,  $A_m$  is a game module  $(N_m, B_m, Y_m, En_m, Ex_m, P_m^0, P_m^1, \delta_m, \eta_m)$  that consists of:

- A finite set of nodes  $N_m$  and a finite set of boxes  $B_m$ .
- A nonempty set of entry nodes  $En_m \subseteq N_m$  and a nonempty set of exit nodes  $Ex_m \subseteq N_m$ .
- A labeling  $Y_m : B_m \rightarrow M$  that assigns a module to every box.
- Let  $Calls_m = \{(b, e) \mid b \in B_m, e \in En_{Y_m(b)}\}$  denote the set of calls of  $m$  and let  $Retns_m = \{(b, x) \mid b \in B_m, x \in Ex_{Y_m(b)}\}$  denote the set of returns in  $m$ . Then,  $\delta_m : N_m \cup Retns_m \rightarrow 2^{N_m \cup Calls_m}$  is a transition function.
- $P_m^0$  and  $P_m^1$  form a partition of  $N_m \cup B_m$  into the set of positions of player 0 and player 1, respectively.
- $\eta_m$  is a labeling function  $\eta_m : N_m \rightarrow \Sigma$  that associates a letter in  $\Sigma$  with each node.

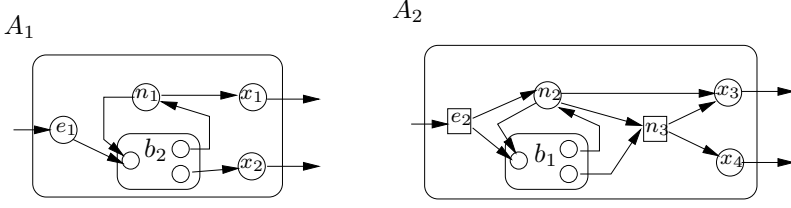
We assume that all the sets  $N_m$  and  $B_m$  ( $m \in M$ ) are disjoint. Also, we let  $N = \bigcup_m N_m$ ,  $B = \bigcup_m B_m$ ,  $Calls = \bigcup_m Calls_m$  and  $Retns = \bigcup_m Retns_m$  denote the set of all nodes, boxes, calls and returns, respectively. Similarly, let  $En = \bigcup_{m \in M} En_m$ ,  $Ex = \bigcup_{m \in M} Ex_m$ ,  $P^\ell = \bigcup_{m \in M} P_m^\ell$ , for  $\ell \in \{0, 1\}$ . We extend the functions  $Y_m$  to a single function  $Y : B \rightarrow M$  by defining  $Y(b) = Y_m(b)$ , where  $b \in B_m$ . Similarly, we extend the functions  $\eta_m$  to a single function  $\eta : N \rightarrow \Sigma$ .

An element in  $Calls_m$  of the form  $(b, e)$  represents a call from  $m$  to the module  $m'$ , where  $Y_m(b) = m'$  and  $e$  is an entry of  $A_{m'}$ . An element in  $Retns_m$  of the form  $(b, x)$  corresponds to the associated return of control from  $m'$  to  $m$  when the call exits from  $m'$  at exit  $x$ . The transition function hence defines moves from nodes and returns to a set of nodes and calls.

We denote the set of *vertices* of  $m$  as  $V_m = N_m \cup Calls_m \cup Retns_m$ . Viewed in terms of vertices, each  $A_m$  defines a graph over the set of vertices  $V_m$ . Let  $V_m^\ell = (N_m \cap P_m^\ell) \cup \{(b, x) \in Retns_m \mid b \in P_m^\ell\}$  denote the set of vertices of player  $\ell$ . Note that returns are identified as belonging to player  $\ell$  if the box corresponding to it belongs to player  $\ell$ . The vertices in  $Calls_m$  are not assigned to any player. (One can assign nodes to players in various ways; we have just chosen one such way, without any real loss in generality.)

Without loss of generality, we make some assumptions of these graphs in the sequel that enable a more readable presentation:

- There is only *one* entry point to every module, i.e.  $|E_m| = 1$  for every  $m$ . We refer to this unique entry point of  $A_m$  as  $e_m$ . One can reduce a game-graph module with multiple entries to this setting by making copies of this module, one for each entry point, and changing the calls and returns appropriately. This causes only a cubic blow-up (see [3] where this is done for reachability specifications).



**Fig. 1.** An example of a recursive game graph.

- For every  $u \in N_m$ ,  $e_m \notin \delta_m(u)$  holds, and for every  $x \in Ex_m$ ,  $\delta_m(x)$  is empty. That is, within a module there are no transitions to its entry point and no transitions from its exits.
- A module is not called immediately after a return from another module. That is, for any  $m$ ,  $\delta_m(Retns_m) \subseteq N_m$ .

As an example of a recursive game graph consider the graph in Figure 1. There are two modules  $A_1$  and  $A_2$ , and  $A_1$  is the initial module. We denote by squares the nodes of player 1 and by circles those of player 0. The rectangles with curved corners denote the boxes of player 0. Box  $b_1$  is mapped by  $Y$  to module  $A_1$  and  $b_2$  is mapped to  $A_2$ . In this recursive game graph, all boxes belong to player 0.

A *state* of the game is an element in  $(\gamma, u) \in B^* \times N$  such that either

- $\gamma = \epsilon$  and  $u \in N_{min}$ , or,
- $\gamma = b_1 \dots b_k$ ,  $b_1 \in B_{m_{in}}$  and for each  $i \in [1, k-1]$ , if  $Y(b_i) = m$ , then  $b_{i+1} \in B_m$ , and  $u \in N_{m'}$  where  $Y(b_k) = m'$ .

Intuitively a state of the form  $(b_1 b_2, u)$  represents the global state of the system where, if  $Y(b_1) = m$  and  $Y(b_2) = m'$ , then the module  $A_{min}$  has called module  $A_m$  by box  $b_1$ , which in turn has called module  $m'$  by box  $b_2$  and the current node inside  $m'$  is  $u$ . Hence, in a state  $(\gamma, u)$ ,  $\gamma$  denotes the *stack* of calls and  $u$  denotes the current node in the last invoked module.

The following then defines the natural notion of a run of an RGG. A *run* of an RGG  $A$  is a finite or infinite non-null sequence of states,  $s_0 s_1 \dots$  such that:

1.  $s_0 = (\epsilon, e_0)$
2. If  $s_j = (\gamma, u)$ ,  $u \in N_m$ , and  $s_{j+1} = (\gamma', u')$ , then one of the following holds:

**Internal move:**  $u \in N_m \setminus Ex_m$ ,  $u' \in \delta_m(u)$  and  $\gamma' = \gamma$ .

**Call a module:**  $u \in N_m \setminus Ex_m$ ,  $(b, e_{m'}) \in \delta_m(u)$ ,  $u' = e_{m'}$  and  $\gamma' = \gamma.b$ .

**Return from a call:**  $u \in Ex_m$ ,  $\gamma = \gamma'.b$ ,  $u' \in \delta_{m'}((b, u))$ , where  $b \in B_{m'}$ .

The first case above is when the control stays within module  $m$ , the second case is when a new module  $m'$  is entered via a box of  $m$  (and we “push” the box name  $b$  onto the stack) and the third is when the control exits  $m$  and returns to  $m'$  (and we “pop” the box name  $b$  from the stack). A *play* in  $A$  is a run of  $A$ . We denote by  $\Pi_f$  and  $\Pi_\omega$  the set of all finite and infinite plays of  $A$ , respectively.

For a state  $s = (\gamma, u)$ , let  $V(s)$  denote the vertex corresponding to the state:  $V(s) = u$  if  $\gamma = \epsilon$  or  $u \in N_m \setminus Ex_m$ ; otherwise  $V(s) = (b, u)$  where  $\gamma = \gamma'.b$ . Then  $ctr : \Pi_f \rightarrow M$  identifies the module where the control is after any finite run and is defined as follows: for any  $\pi.s \in \Pi_f$ ,  $ctr(\pi.s) = m$ , where  $V(s) \in V_m$ .

We can now define local histories that describe, for every finite play  $\pi$  such that  $\text{ctr}(\pi) = m$ , the history of the play within  $m$  since the current invocation of the module  $m$ . For a state  $s = (b_1 \dots b_r, u)$ , an  $s$ -history is  $\langle \beta_1, \beta_2, \dots, \beta_r \rangle$ , where for each  $i \in [r]$ ,  $\beta_i \in V_m^*$  where  $b_i \in B_m$ . In other words, each  $\beta_i$  is a sequence of vertices of the module  $b_i$  belongs to. We define a *history* function  $Hst$  that associates with every finite play  $\pi.s$  an  $s$ -history  $Hst(\pi.s)$  as follows.

- If  $\pi = s_0 = (\epsilon, e_{m_{in}})$ , then  $Hst(\pi) = \langle e_{m_{in}} \rangle$ .
- Let  $\pi = \pi'.(\gamma', u')$  where  $\pi' = \pi''.(\gamma, u)$ , and let  $Hst(\pi') = \langle \beta_1, \dots, \beta_r \rangle$ . Then:

**Internal move:** If  $\gamma' = \gamma$ , then  $Hst(\pi) = \langle \beta_1, \dots, \beta_{r-1}, \beta_r.u' \rangle$ .

**Call:** If  $\gamma' = \gamma.b$  with  $Y(b) = m'$ , then  $Hst(\pi) = \langle \beta_1, \dots, \beta_r, e_{m'} \rangle$ .

**Return:** If  $\gamma = \gamma'.b$ , then  $Hst(\pi) = \langle \beta_1, \dots, \beta_{r-1} \rangle$ .

Intuitively, in the beginning of the run, when the control is at  $m_{in}$ , the history is a single string  $e_{in}$ . On an internal move, the last element in the history tuple gets updated, while the other tuples remain unchanged. On a call, the last element in the history tuple also freezes, and a new entry is created and initialized to the entry corresponding to the called module. When the call returns, the last tuple in the history gets erased. The history function hence keeps a *stack* of histories: for each module in the stack of calls, there is a record of the moves that have happened in that module during the play thus far.

We now define the *local history*,  $\mu$ , after a finite play  $\pi$ . For any play  $\pi$ ,  $\mu(\pi) = \beta_k$  where  $Hst(\pi) = \langle \beta_1, \dots, \beta_k \rangle$ . In other words, the local history is the fragment of the play inside the current module  $m = \text{ctr}(\pi)$  since the corresponding entry into  $m$ . Note that the local history is a sequence of vertices that correspond to internal nodes, calls or returns of the module  $m$ .

A *modular strategy* is intuitively a set of functions, one for each module, that encodes how player 0 must play in the game. However, which move to make at a state can depend only on the *local history* of the play so far in the current module. Formally, a modular strategy is a set of functions,  $f = \{f_m\}_{m \in M}$ , one for each module, where for every  $m$ ,  $f_m : V_m^* \cdot V_m^0 \rightarrow V_m$  such that  $f_m(\pi v) \in \delta_m(v)$ , for every  $\pi \in V_m^*$ ,  $v \in V_m^0$ .

A play  $\pi$  in  $A$  according to a modular strategy  $f$  is a run  $s_0 s_1 \dots$  such that for every  $i < |\pi|$ , if  $\text{ctr}(s_0 \dots s_i) = m$  and  $V(s_i) \in V_m^0$ , then  $s_{i+1} = (\gamma', u')$ , where  $f_m(\mu(s_0 \dots s_i))$  is either  $u'$  or  $(b, u')$ , for some  $b \in B$ . In other words, if a prefix of the play ends in a player 0 vertex, the move recommended by  $f$  for the local memory in the current module must be taken.

**Winning conditions.** A *winning set* over  $\Sigma$  is a regular  $\omega$ -language over  $\Sigma$ , i.e. a regular language  $\mathcal{L} \subseteq \Sigma^\omega$ . A *recursive game* is a pair  $(A, \mathcal{L})$  where  $A$  is a recursive game graph and  $\mathcal{L}$  is a winning set, both over  $\Sigma$ . Let us extend  $\eta$  to states by defining  $\eta((\gamma, u)) = \eta(u)$ . This then extends to plays:  $\eta(s_0 s_1 \dots) = \eta(s_0)\eta(s_1) \dots$ . A play  $\pi$  is said to be *winning* if  $\eta(\pi) \in \mathcal{L}$ , the winning set. A modular strategy is *winning* if every play according to it is winning. We consider the following decision problem:

“Given a recursive game  $(A, \mathcal{L})$  is there a modular winning strategy for the protagonist?”

In this paper we solve this problem for  $\mathcal{L}$  given by *safety* and *deterministic/universal Büchi/co-Büchi* automata, and by LTL formulas.

### 3 Automata for Winning Strategies

#### 3.1 Background: Words, Trees and Automata

An automaton on  $\omega$ -words over  $\Sigma$  is  $\mathcal{A} = (Q, q_1, \delta, W)$  where  $Q$  is a finite set of states,  $q_1 \in Q$  is the initial state,  $\delta : Q \times \Sigma \rightarrow 2^Q$  is a transition function and  $W$  is a *winning condition*. Depending on the winning condition we have a Büchi, co-Büchi, or a parity automaton. A *Büchi* (resp. *co-Büchi*) winning condition is  $W \subseteq Q$  and requires that a state from  $W$  repeats infinitely often (resp. only finitely often). A *parity* condition is a function  $W : Q \rightarrow [c]$ , for some  $c \in \mathbb{N}$ , that associates a colour to every element in  $Q$ . It requires that the minimal colour seen infinitely often is even. A run of  $\mathcal{A}$  over a word  $\sigma_0\sigma_1 \dots \in \Sigma^\omega$  is an  $\omega$ -sequence of states  $p_0p_1 \dots$  such that  $p_0 = q_1$  and  $\forall i \in \mathbb{N}, p_{i+1} \in \delta(p_i, \sigma_i)$ . A run is accepting if it satisfies the winning condition  $W$ . An automaton is *deterministic* if  $|\delta(q, \sigma)| \leq 1$  for every  $q \in Q, \sigma \in \Sigma$ . For either deterministic or nondeterministic automata a word  $\alpha \in \Sigma^\omega$  is accepted if *there exists* an accepting run on it. For *universal* automata  $\alpha$  is accepted if *all* runs on it are accepting. A *safety* automaton is a deterministic Büchi automaton  $\mathcal{A} = (Q, q_1, \delta, W)$  where  $W = Q$ . In other words, it accepts a word as long as it has a run over the word, thus we omit mentioning the winning condition.

Let  $k \in \mathbb{N}$  and  $\Delta$  be a finite alphabet. A  $\Delta$ -labelled  $k$ -tree is  $(T_k, \nu)$  where  $T_k = (Z, E)$  is a tree with  $Z = [k]^*$  and  $E = \{(y, y.d) \mid y \in [k]^*, d \in [k]\}$ , and  $\nu : Z \rightarrow \Delta$  is a labelling function that labels every vertex of the tree with a letter in  $\Delta$ . To distinguish vertices of trees and vertices of recursive game graphs, we refer to the former as *tree-vertices*. The tree-vertex  $\epsilon$  is the *root* of the tree  $T_k$ , and for every  $y \in Z$ , tree-vertex  $y.d$  is the  $d^{\text{th}}$  child of  $y$ . For any set  $X$ , let  $\mathcal{B}^+(X)$  denote the set of boolean formulae over  $X$  using conjunctions and disjunctions only (negation is not allowed). For any subset  $F$  of  $X$ , we say that  $F$  *satisfies* a formula  $\varphi \in \mathcal{B}^+(X)$  if  $\varphi$  evaluates to true assigning the elements in  $F$  to *true* and the other elements in  $X$  to *false*.

A *two-way alternating parity tree automaton* (see [18]) over  $\Delta$ -labelled  $k$ -trees is  $\mathcal{A} = (Q, q_1, \delta, W)$ , where  $Q$  is a finite set of states,  $q_1 \in Q$  is the initial state,  $W$  is a parity condition on  $Q$  and  $\delta : Q \times \Delta \rightarrow \mathcal{B}^+((\{-1, 0, 1, \dots, k\} \times Q))$ . Intuitively,  $\{-1, 0, \dots, k\}$  code the directions from a tree-vertex, where  $\{1, \dots, k\}$  stand for the  $k$  children of the tree vertex,  $-1$  stands for the parent of the tree vertex, and  $0$  stands for the current tree-vertex itself. Let us extend the definition of concatenation of words over  $[k]^*$  as follows:  $(xi.(-1)) = x$  and  $x.0 = x$ , for any  $x \in [k]^*, i \in [k]$ , i.e. when a word is concatenated with  $-1$ , it removes the last letter and concatenating with  $0$  is the identity function. A *one-way nondeterministic tree automaton* can be seen as a two-way alternating tree automaton where the transition function is always a disjunction of formulas of the kind  $\bigwedge_{j=1}^k (j, q_j)$ , i.e. the automaton guesses nondeterministically to send exactly *one* copy of itself in each *forward* direction.

A run of  $\mathcal{A}$  over a  $\Delta$ -labelled  $k$ -tree  $(T_k, \nu)$ , where  $T_k = (Z, E)$ , is a labelled tree  $T_\rho = (R_\rho, E_\rho)$  where each tree-vertex in  $R_\rho$  is labelled with a pair  $(x, q)$  where  $x \in Z$  is a tree-vertex of the input tree and  $q \in Q$  is a state of the automa-

ton  $\mathcal{A}$ , such that: (a) the root of  $T_\rho$  is labelled  $(\epsilon, q_1)$ , and (b) if a tree-vertex  $y$  of  $T_\rho$  is labelled  $(x, q)$ , then we require that there is a set  $F \subseteq \{-1, 0, 1, \dots, k\} \times Q$  such that  $F$  satisfies  $\delta(q, \nu(x))$  and for each  $(i, q') \in F$ ,  $y$  has a child labelled  $(x.i, q')$ . A run is *accepting*, if for every *infinite* path in the run tree, if one projects the second component of the labels along the path, then it is a sequence of states in  $Q$  that satisfies the winning condition of  $\mathcal{A}$ . Note that there is no condition for *finite* paths of the run tree. An automaton  $\mathcal{A}$  accepts a  $\Delta$ -labelled  $k$ -tree  $T$  iff there is an accepting run of  $\mathcal{A}$  on  $T$ ; the language of  $\mathcal{A}$ , denoted  $\mathcal{L}(\mathcal{A})$ , is the set of all  $\Delta$ -labelled  $k$ -trees that  $\mathcal{A}$  accepts.

**Proposition 1.**

- Let  $\mathcal{A}$  be a two-way alternating parity tree automaton. Then there is a one-way nondeterministic parity tree automaton  $\mathcal{A}'$  such that  $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$ , the number of states in  $\mathcal{A}'$  is exponential in the number of states in  $\mathcal{A}$ , and the number of colours in the parity condition of  $\mathcal{A}'$  is linear in the number of colours in the parity condition of  $\mathcal{A}$  [18].
- The emptiness of one-way parity tree automata can be checked in time that is polynomial in the number of states and exponential in the number of colours in the parity condition (see [17]).

### 3.2 Strategy Trees

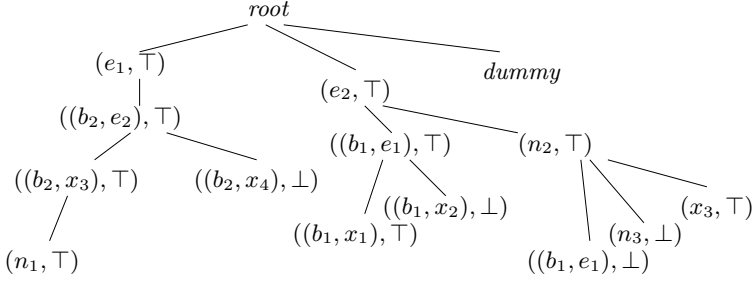
Let us fix a set of  $h$  modules  $M = \{m_1, \dots, m_h\}$ , which is ordered, and a finite alphabet  $\Sigma$  for this section. Let us fix a recursive game graph over  $\Sigma$ ,  $A = (M, m_1, \{A_m\}_{m \in M})$ , where each  $A_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m)$ . By  $\Delta_A$  we denote the set of labels  $\{root, dummy\} \cup (V \times \{\top, \perp\})$ .

A *strategy tree* is a  $\Delta_A$ -labelled  $k$ -tree and is intended to encode a modular strategy in the following way. First, we have the special symbol *root* labelling the root of the tree. The subtree rooted at the  $i^{th}$  child of the root will encode the modular strategy for  $m_i$ . The root of the subtree for  $m$  is labelled by the entry point of  $m$  and tagged with  $\top$ . (The other children of the root, if any, are marked to be dummy tree-vertices and will not encode any information). No other tree-vertices are labelled from the set  $En \times \{\top, \perp\}$ .

The subtree for a module  $m$  encodes the strategy for the module  $m$  by unravelling the graph  $A_m$  and annotating each tree-vertex with either  $\top$  or  $\perp$ , with  $\top$  intuitively encoding that a move to the corresponding node is possible while a  $\perp$ -tag signifies that the strategy will not allow a play to visit this node.

If a tree-vertex  $v$  of the subtree for  $m$  is labelled  $(p, \top)$ , then:

- When  $p \in (N_m \setminus Ex_m) \cup Retns_m$ , the children of  $v$  are labelled by the successors of  $p$  along with a  $\top/\perp$  annotation. Further, if  $p$  is a player 0 vertex, then the strategy must choose exactly one successor of  $p$ , and thus exactly one of the annotations of the children is  $\top$ . If  $p$  is a player 1 vertex, then *all* moves of player 1 need to be accounted for, hence all children are tagged with  $\top$ .
- When  $p \in Calls_m$ , we *do not* encode calling the other module; we instead simply have children corresponding to the returns from the called module, with any  $\top/\perp$  annotation. This hence encodes an assumption that the call to



**Fig. 2.** A fragment of a strategy tree.

the other module will definitely not end in any return which is tagged with  $\perp$ , no matter how player 1 plays. Note that we do not put any constraints at this point that this assumption is indeed true—we take care of this later when we check these trees using automata.

If a tree-vertex of the subtree for  $m$  is labelled  $(p, \perp)$ , this denotes a move disabled by the strategy and we do not continue to define the strategy from it; we hence label the entire subtree under it as *dummy*. Similarly, if a tree-vertex is labelled  $(p, \zeta)$  where  $p \in Ex_m$ , then this signifies reaching an exit node of the module and again we do not encode the strategy after this point.

For example, consider the recursive game graph in Figure 1, and a modular strategy such that the choices for player 0 are resolved selecting  $(b_2, e_2)$  from  $n_1$  and  $x_3$  from  $n_2$ . A fragment of the corresponding strategy tree is shown in Figure 2. Note that the subtree encoding the strategy for module  $A_1$  assumes that when  $A_2$  is called, it will not exit at  $x_4$ , and the subtree for  $A_2$  assumes that when  $A_1$  is called, it will not exit at  $x_2$ . In this example, the strategy does indeed guarantee these assumptions. It is easy to see that:

**Proposition 2.** *There exists an effectively constructible one-way nondeterministic tree automaton  $\mathcal{A}_{str}$  of size  $O(|\mathcal{A}|)$  that accepts a  $\Delta$ -labelled  $k$ -tree if and only if it is a strategy tree.*

### 3.3 Automata Accepting Winning Strategy Trees

In this section we sketch the construction of an automaton  $\mathcal{A}_{win}$  that accepts a strategy tree iff it represents a winning strategy with respect to a specification given as a safety automaton  $\mathcal{B} = (Q, q_1, \delta_{\mathcal{B}})$ . In this construction it is crucial the use of an *avoid component* in states of  $\mathcal{A}_{win}$ . An avoid component  $(x, q) \in Ex \times Q$  corresponds to the assumption that a play must not end at the exit  $x$  with the specification state  $q$ . There is also another type of avoid component denoted  $\$m$  that is used instead for the assumption that the play must not exit at all the current invocation of the module  $m$ . We will return to the avoid component later.



Automaton  $\mathcal{A}_{win}$  is a two-way alternating tree automaton that performs three main tasks. The first task is to simulate  $\mathcal{B}$  so that we ensure the satisfaction of the specification on an accepted strategy tree. This is achieved basically simulating the transitions from  $\delta_{\mathcal{B}}$ , whenever a node is read. On reading an exit node the automaton also checks that the assumption of the avoid component is fulfilled.

The second task is to ensure that plays are generated from the entry point of the initial module  $m_{in}$  starting with the initial state of the specification, and that this invocation of  $m_{in}$  does not exit. If this call exits then this would mean that there are finite maximal plays according to the strategy, which are by definition losing for player 0. Hence, the first transition sets the avoid component to every possible pair  $(x, q) \in Ex_{m_{in}} \times Q$ .

The last task is to ensure that when a return of the input tree is tagged with  $\perp$ , no matter how player 1 plays, the call to the other module will definitely not end in such a return (see Section 3.2). Assuming that this return is from a module  $m$  at exit  $x$ , the above requirement is checked sending a copy of the automaton to the root of the subtree corresponding to  $m$  for *every* avoid component of the form  $(x, q)$ .

The winning condition we choose for  $\mathcal{A}_{win}$  is trivial: we just map every state to the colour 0; hence the winning condition simply requires that there be a run over the tree. There is a trick that is needed in the construction to keep the size of  $\mathcal{A}_{win}$  small. When reading a  $\top$ -tagged return  $(b, x)$  ( $x \in Ex_{m'}$ ), the strategy encodes the assumption that when  $m'$  is called, it may return at exit  $x$ , but it does not contain information on what the specification state can be when it returns. The automaton hence will guess what the set of possible specification states can be when the call returns. Since we would like to keep the size of the transition function polynomial, we let the automaton step through the specification states  $q$  one by one, but staying at the same tree vertex, and guess whether the current specification state  $q$  is a possibility as the state on a return. We omit further details here. Thus, we have the following.

**Lemma 1.**  *$\mathcal{A}_{win}$  is a two-way alternating tree automaton that accepts a strategy tree iff it corresponds to a winning strategy. Further, the size of  $\mathcal{A}_{win}$  is  $O(|Q|^2 \cdot |Ex|)$  where  $Q$  is the set of states in the specification automaton and  $Ex$  is the set of all exits in the recursive game graph.*

We now convert  $\mathcal{A}_{win}$  to a one-way nondeterministic tree automaton and take its intersection with the automaton  $\mathcal{A}_{str}$  that accepts strategy trees, to get a one-way nondeterministic automaton  $\mathcal{A}'$  which accepts a tree iff it corresponds to a winning strategy tree. Since we can prove EXPTIME-hardness, via a direct reduction from linear-space Turing machines, and using Proposition 1, we have:

**Theorem 1.** *For a recursive game graph  $A$  and a specification safety automaton  $\mathcal{B}$ , the problem whether there is a winning modular strategy for player 0 is EXPTIME-complete, and can be decided in time  $O(|A| \cdot \exp(|Ex| \cdot |\mathcal{B}|))$ .<sup>1</sup>*

<sup>1</sup>  $\exp(x)$  stands for  $2^x$ .



## 4 Complexity of Modular Games

### 4.1 Handling Other $\omega$ -Regular Specifications

The construction given in Section 3.3 can be extended to games with specifications given as deterministic Büchi or co-Büchi automata on words. We consider here only the case of Büchi automata; the case for co-Büchi automata is dual.

Let  $\mathcal{B} = (Q, q_1, \delta_{\mathcal{B}}, W)$  be a deterministic Büchi automaton on words, and  $A$  be a recursive game graph. We can extend the automaton  $A_{win}$  of Section 3.3 to a two-way alternating tree automaton  $\mathcal{A}'_{win}$  that accepts strategy trees corresponding to winning strategies in this game.

The main modifications are as follows. When the automaton reads a call in the subtree for a module  $m$ , recall that we guessed for every return labelled  $\top$ , the set of specification states the play could be at when it exits from the called module. In the case of Büchi specifications, we need to guess more about what happened during the call. More precisely, we need to know whether *all* possible sub-plays that return at this exit and specification state would have definitely met a state in  $W$  or not. If the automaton guesses that this is so, it must send a copy to the called module to check this, and also signal a Büchi final state (for the tree automaton) before continuing the play in the current module.

Again, by Proposition 1, and the lower bound given in Theorem 1, we have.

**Theorem 2.** *Deciding recursive game graphs with deterministic Büchi (or co-Büchi) automaton specifications is EXPTIME-complete.*

We can also extend the tree automaton from Section 3.3 to handle specifications given as *universal* Büchi (or co-Büchi) specifications. In the construction of the automaton, whenever we were updating the specification state, we now need to create a copy for *each possible update* of the specification state.

**Theorem 3.** *Deciding recursive game graphs with universal Büchi (or co-Büchi) automaton specifications is EXPTIME-complete.*

The above theorem in fact shows that we can solve games for any  $\omega$ -regular specification. If  $\mathcal{L}$  is any  $\omega$ -regular language, then its complement,  $\bar{\mathcal{L}}$  is also  $\omega$ -regular and can be accepted by some nondeterministic Büchi automaton  $\mathcal{B}'$ . It is easy to see that if  $\mathcal{B}'$  is viewed as a *universal co-Büchi automaton*  $\mathcal{B}$  (we keep the same states and the same transitions but interpret the automaton as universal and co-Büchi), then  $\mathcal{B}$  accepts a word  $\alpha$  iff  $\mathcal{B}'$  rejects  $\alpha$ . Hence  $\mathcal{B}$  accepts  $\mathcal{L}$  and we can solve games against this automaton.

We can also deal with LTL-specifications ([15]) over a set of propositions  $\mathcal{P}$ , assuming the nodes in the game graph are labelled over  $\Sigma = 2^{\mathcal{P}}$ . Given an LTL formula  $\phi$ , we can construct a nondeterministic Büchi automaton  $B_{\neg\phi}$  over  $2^{\mathcal{P}}$  that accepts a word iff it satisfies  $\neg\phi$  [19]. Moreover, the size of this automaton is exponential in  $|\phi|$ . By the previous observation, this automaton when viewed as a universal co-Büchi automaton accepts the models of  $\phi$ . In other words, for any LTL formula, we can construct an exponential sized universal co-Büchi automaton accepting its models. Invoking Theorem 3 and using the fact that LTL-games are 2EXPTIME-hard [16] even for normal graphs, we have:

**Theorem 4.** *Deciding LTL recursive game graphs is 2EXPTIME-complete.*

## 4.2 Structure Complexity

Our algorithm for deterministic automata specifications works in time exponential in the number of exits in the recursive game graph. We could ask whether this exponential is necessary by asking for the *structural complexity* of the problem, i.e. what is the complexity for fixed  $\omega$ -regular specifications. Since a reachability game on a recursive game graph can be formulated by a simple fixed automaton specification, and since reachability games in recursive game graphs are NP-hard [3], we cannot hope to do polynomial in the game graph. However, we can show that for fixed specifications, the problem is in NP.

The NP upper bound can be shown by direct means, not involving automata. Let us sketch the proof for deterministic safety automata. When a module  $m$  invokes a module  $m'$  at a specification state  $q$ , the only relevant information that it needs is the set of exits  $m'$  could return at, and at each of these exits, the possible states the specification automaton can be in. This is a polynomial-sized information that we can guess. For each module  $m$  and specification state  $q$ , we build a graph  $G_{m,q}$  which is a product of the game module for  $m$  and the specification automaton starting at state  $q$ . The problem then boils down to finding whether, for every module  $m$ , there is a strategy for the game module for  $m$  such that, when this strategy is played on  $G_{m,q}$ , for every  $q \in Q$  (in the obvious way), the strategy meets the assumption pertaining to how module  $m$  should behave when invoked at  $q$ . In doing this, whenever we call another module in  $G_{m,q}$ , we can “plug-in” a graph that captures the assumption on how the other modules will behave. For any  $m$ , solving the game graphs  $G_{m,q}$  simultaneously is akin to solving partial information games, which causes an exponential blow-up only in the size of the specification state space  $Q$ , which is a constant. Hence the problem can be solved in NP. The procedure extends to universal Büchi and co-Büchi automata with some effort.

**Theorem 5.** *Deciding recursive game graphs for fixed  $\omega$ -regular specifications is NP-complete.*

## 5 Conclusions

In this paper, we have solved the problem of deciding the existence of modular strategies in infinite games over recursive structures for winning conditions specified using  $\omega$ -automata or linear temporal logic. We have argued that the notion of modular strategies, compared to the traditional definition of global strategies, is more appropriate for designing modules that can be plugged in any context. Our solution is automata theoretic, and can be generalized to allow other types of specifications such as branching time logics. In terms of future work, we are exploring the application of games to generate interface abstractions, and efficient implementations using a combination of BDD-based symbolic techniques and SAT solvers.

## References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. 13th Intern. Conf. on Computer Aided Verification, CAV'01*, LNCS 2102, p. 207–220. Springer, 2001.
2. R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.
3. R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *Proc. 9th Intern. Conf. on Tools and Algorithms for the Construction and the Analysis of Systems, TACAS'03*, LNCS 2619, p. 363–378. Springer, 2003.
4. T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. 13th Intern. Conf. on Computer Aided Verification, CAV'01*, p. 260–264, 2001.
5. M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. 28th Intern. Coll. on Automata, Languages and Programming, ICALP'01*, LNCS 2076, p. 652–666. Springer, 2001.
6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. 8th Conference on Concurrency Theory, CONCUR'97*, LNCS 1243, p. 135–150, 1997. Springer, 1997.
7. J. Büchi and L. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:295 – 311, 1969.
8. T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proc. 29th Intern. Coll. on Automata, Languages and Programming, ICALP'02*, LNCS 2380, p. 704–715. Springer, 2002.
9. T. Cachat. *Two-Way Tree Automata Solving Pushdown Games*, p. 303–317. LNCS 2500. Springer, 2002.
10. A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *Proc. 14th Intern. Conf. on Computer Aided Verification, CAV '02*, LNCS 2404, p. 428–441. Springer, 2002.
11. H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. 9th ACM Conf. on Comp. and Comm. Security*, 2002.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. 12th Intern. Conf. on Computer Aided Verification, CAV '00*, LNCS 1855, p. 232–247. Springer, 2000.
13. O. Kupferman and M. Vardi. An automata-theoretic approach to reasoning about infinite-state systems. In *Proc. 12th Intern. Conf. on Computer Aided Verification, CAV '00*, LNCS 1855, p. 36–52. Springer, 2000.
14. O. Kupferman, M. Vardi, and P. Wolper. Module checking. *Information and Computation*, 164(2):322–344, 2001.
15. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, p. 46 – 77, 1977.
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.
17. W. Thomas. Infinite games and verification. In *Proc. 14th Intern. Conf. on Computer Aided Verification, CAV'02*, LNCS 2404, p. 58–64. Springer, 2002.
18. M. Vardi. Reasoning about the past with two-way automata. In *Proc. 25th Intern. Coll. on Automata, Languages, and Programming, ICALP'98*, LNCS 1443, p. 628–641. Springer, 1998.
19. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1 – 37, 1994.
20. I. Walukiewicz. Pushdown processes: Games and model-checking. *Information and Computation*, 164(2):234–263, January 2001.

# Fast Mu-Calculus Model Checking when Tree-Width Is Bounded

Jan Obdržálek

Laboratory for Foundations of Computer Science  
The University of Edinburgh  
j.obdrzalek@ed.ac.uk

**Abstract.** We show that the model checking problem for  $\mu$ -calculus on graphs of bounded tree-width can be solved in time linear in the size of the system. The result is presented by first showing a related result: the winner in a parity game on a graph of bounded tree-width can be decided in polynomial time. The given algorithm is then modified to obtain a new algorithm for  $\mu$ -calculus model checking. One possible use of this algorithm may be software verification, since control flow graphs of programs written in high-level languages are usually of bounded tree-width. Finally, we discuss some implications and future work.

## 1 Introduction

The modal  $\mu$ -calculus introduced by Kozen [10] is a very expressive fixpoint logic capable of specifying a wide range of properties of (non-terminating) programs, such as safety, liveness, fairness etc. Moreover, many important temporal logics were shown to be fragments of the modal  $\mu$ -calculus [4,5].

Even though modal  $\mu$ -calculus was extensively studied, the exact complexity of model checking problem for this logic is not known. The original result of Emerson and Lei [5] states the following: *The model checking problem for a formula of size  $m$  and alternation depth  $d$  on a system of size  $n$  is  $\mathcal{O}(m \cdot n^{d+1})$ .* So model checking is exponential in alternation depth of the formula. In [4] the complexity was shown to be in  $\text{NP} \cap \text{co-NP}$ , though it is unlikely for the problem to be NP-complete. A substantial effort was undertaken to find a polynomial model checking algorithm. The only improvement since the original work came from Long et al. [11]. Their delicate result allowed the complexity to be decreased to  $\mathcal{O}(m \cdot n^{\lceil d/2 \rceil + 1})$ .

In contrast with the explicit computation of fixpoints in the original algorithm [5], the paper [4] shows that the model checking problem for  $\mu$ -calculus is equivalent to the non-emptiness problem for automata on infinite trees with parity acceptance condition. As a consequence of this fact it can be shown that this is equivalent to the problem of deciding a winner in parity games (there is a polynomial-time reduction). See also [15]. Parity games were therefore extensively studied [8,18], but so far this research has not come up with a polynomial algorithm.

*Tree-width* is a graph theoretic concept introduced first by Robertson and Seymour [14] in their work on graph minors. Roughly speaking, tree-width measures how close is the given graph to being a tree. Graphs with low tree-width then allow a decomposition of the problem being solved into subproblems, decreasing the overall complexity – some in general NP-complete problems were shown to be polynomial on these graphs. (Following the intuition that solving problems on trees is *much* easier than on general graphs. E.g. modal  $\mu$ -calculus model checking is linear on trees.) See Bodlaender’s paper [1] for an excellent survey.

Even though the concept of tree-width is quite restrictive, in practice the systems considered are surprisingly often of a low tree-width. In [17] it was shown that all C programs (resp. their control-flow graphs) are of tree-width at most 6, and Pascal programs of tree-width at most 3! This result does not hold for Java, as the labelled versions of `break` and `continue` can be as harmful as `goto` [7]. In practice, however, programs with high tree-width do not appear (since they are written by sane humans).

In this paper we show that for graphs of bounded tree-width, the  $\mu$ -calculus model checking problem can be solved in time  $\mathcal{O}(n \cdot \alpha(m, k))$  on systems of tree-width  $k$ . In general this is a consequence of a general theorem of Courcelle [2]: For a fixed MSO formula  $\varphi$  and a graph of bounded tree-width the model checking problem can be solved in time linear in the size of the graph. This result, however, does not provide any estimate on  $\alpha(m, k)$  (except for being “large”). Recently it was shown [6] that  $\alpha$  is not even elementary. Moreover the algorithm itself is quite complicated and does not provide any insight into what are the results/strategies in the underlying game.

In contrast, our algorithm does not require translation into MSO. Its complexity is clearly expressed in the parameters  $m, k$  and  $d$  and in addition one can easily follow the workings of the algorithm as well as the evolving strategies. We start by first proving a related important result: That a winner in a parity game can be determined in polynomial time (linear when the number of colors is fixed). This result is new and does not follow from [2]. We then extend this to give a new  $\mu$ -calculus model checking algorithm.

**Acknowledgement.** I am indebted to Colin Stirling for his invaluable support, guidance, and for suggesting me to work on this topic in the first place.

## 2 Parity Games and $\mu$ -Calculus

### 2.1 Parity Games

The *Parity game*  $\mathcal{G} = (V_0, V_1, E, \lambda)$  consists of a directed graph  $D = (V, E)$ , where  $V$  is disjoint union of  $V_0$  and  $V_1$ , and a parity function  $\lambda : V \rightarrow \mathbb{N}$  ( $0 \notin \mathbb{N}$ ). For clarity we assume that for every vertex of  $V$  there is at least one outgoing edge in  $D$ .

The game is played by two players  $P_0$  and  $P_1$  (called also *Even* and *Odd*), who form an infinite path in  $D$  by moving a token along the edges. The game starts in an initial vertex and players play indefinitely in the following way. If the

token is on vertex  $v$  of  $V_0$ , then player  $P_0$  moves the token along some edge with tail  $v$ . If the token is on vertex of  $V_1$ , player  $P_1$  moves the token. As a result, the players form an infinite path  $\pi : \pi_1 \pi_2 \dots$ , which corresponds to the infinite sequence of priorities  $\text{Inf}(\pi) : p(\pi_1)p(\pi_2)\dots$ . Player  $P_1$  wins the path (play)  $\pi$  if the highest priority appearing infinitely often in  $\text{Inf}(\pi)$  is odd, otherwise player  $P_0$  wins.

Next we define the notion of strategy for player  $P_0$  (it is dual for player  $P_1$ ). A (memoryless) *strategy*  $S$  assigns to every vertex  $v \in V_0$  one of the edges of  $E$  with a tail in  $v$ . More formally,  $S$  is a function  $S : V_0 \rightarrow V$  s.t.  $\forall v \in V_0. S(v) = w \implies (v, w) \in E$ . A player plays *using a strategy*  $S$ , if in his vertex  $v$  he always chooses  $S(v)$  as the next vertex. We say that a strategy  $S$  is winning for player  $P_0$  (resp. player  $P_1$ ) from  $v \in V$ , if he wins every play starting in  $v$  using this strategy.

From the well known determinacy result [3] we know that the vertex set  $V$  can be divided into two sets  $W_0$  and  $W_1$ , containing the vertices for which the player  $P_0$  ( $P_1$ ) has a winning strategy. Moreover, it is shown sufficient to take just memoryless strategies defined as above. Every strategy for player  $P_0$  gives us a graph  $D^S$ :

**Definition 1** ( $D^S$ ). *For a graph  $D = (V, E)$  and a strategy  $S$  we define  $D^S = (V, E')$  to be a subgraph of  $D$ , where  $E' = E \setminus \{(v, w) \in E \mid v \in V_0 \wedge S(v) \neq w\}$ . In other words, we remove all the edges leaving the vertex  $v$  of  $V_0$  except for the one corresponding to the strategy  $S$ .*

It is easy to show, that on this graph  $P_0$  wins if there is no cycle reachable from the initial vertex such that  $P_1$  wins this cycle.

## 2.2 Modal $\mu$ -Calculus

The syntax of modal  $\mu$ -calculus we use (positive normal form) is defined by:

$$\varphi ::= Z \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi \mid [a] \varphi \mid \mu Z. \varphi \mid \nu Z. \varphi$$

$Z$  here ranges over a family of propositional variables. We do not give the semantics here, as we assume the reader is familiar with the modal  $\mu$ -calculus (see [16] for a good introduction). In the text we also deal only with closed formulas (i.e. not containing any free variables).

We evaluate  $\mu$ -calculus formulas on labelled transition systems (LTSs), an LTS  $\mathcal{T}$  being a triple  $(P, \text{Act}, \longrightarrow)$ . These LTS can for example represent a control flow graph of some program. In the rest of this text we use  $p, q, \dots$  to denote states (members of  $P$ ) and write  $p \xrightarrow{a} q$  for  $(p, a, q) \in \longrightarrow$ .

There is a well known algorithm for constructing a parity game  $\mathcal{G} = (V_0, V_1, E, \lambda)$  corresponding to model checking problem for  $\mathcal{T} = (P, \text{Act}, \longrightarrow)$  and  $\varphi$ . This construction is basically by computing a synchronised product of the graphs of  $\varphi$  and  $\mathcal{G}$  (see eg. [16]). The following theorem holds:

**Theorem 1** ([16]). *Player  $V_0$  has a winning strategy for  $(p, \varphi)$  in the parity game  $\mathcal{G}$  iff  $p \models_{\mathcal{T}} \varphi$ .*

### 3 Tree Decompositions

Here we present the relevant facts about tree decompositions and tree-width, which will be needed later in the text.

**Definition 2 (Tree decomposition).** A tree decomposition of an (undirected) graph  $G = (V, E)$  is a pair  $(\mathcal{X}, T)$ , where  $T = (I, F)$  is a tree (its vertices are called nodes throughout this paper) and  $\mathcal{X} = \{X_i \mid i \in I\}$  family of subsets of  $V$  such that:

- $\bigcup_{i \in I} X_i = V$ ,
- for every edge  $\{v, w\} \in E$  there exists an  $i \in I$  s.t.  $\{v, w\} \subseteq X_i$ , and
- for all  $i, j, k \in I$  if  $j$  is on the (unique) path from  $i$  to  $k$  in  $T$ , then  $X_i \cap X_k \subseteq X_j$ .

The width of a tree decomposition  $(\mathcal{X}, T)$  is  $\max_{i \in I} |X_i| - 1$ . The tree-width of a graph  $G$  is the minimum width over all possible tree decompositions of  $G$ . Trees have tree-width one. One obtains an equivalent definition if the third condition is replaced by:

For all  $v \in V$ , the set of nodes  $\{i \in I \mid v \in X_i\}$  forms a connected subtree of  $T$ .

Let  $i$  be a node of  $T$  in a tree decomposition  $(\mathcal{X}, T)$  of  $D$ . Then we use the following notation:

$T_i$  – subtree of  $T$  rooted in  $i$

$V_i$  – vertices of  $D$  appearing in  $T_i$  – i.e.  $V_i = \bigcup_{j \in T_i} X_j$

$D_i$  – subgraph of  $D$  induced by  $V_i$  – i.e.  $V(D_i) = V_i$  and  $E(D_i) = \{(s, t) \in E \mid s, t \in V_i\}$

$D \setminus W$  – subgraph of  $D$  induced by vertices  $V(D) \setminus W$

The following fact about tree decompositions is one of the basic properties of graphs of bounded tree-width, which allows for all the interesting results.

**Fact 1.** Let  $(\mathcal{X}, T)$  be a tree decomposition and  $i$  a node of  $T$ . Then the only vertices in  $V_i$  adjacent to vertices  $V \setminus V_i$  are those belonging to  $X_i$ . In other words,  $X_i$  is an interface between  $D_i$  and the rest of the graph.

The following notion of *nice tree decomposition* allows us to significantly simplify the construction of our algorithm. This choice is justified by Lemma 1.

**Definition 3 (Nice tree decomposition).** Tree decomposition  $(\mathcal{X}, T)$  is called nice tree decomposition, if the following three conditions are satisfied:

1. every node of  $T$  has at most two children,
2. if a node  $i$  has two children  $j$  and  $k$ , then  $X_i = X_j = X_k$ , and
3. if a node  $i$  has one child  $j$ , then either  $|X_i| = |X_j| + 1$  and  $X_j \subseteq X_i$  or  $|X_i| = |X_j| - 1$  and  $X_i \subseteq X_j$ .

**Lemma 1 (See [9]).** *Every graph  $G$  of tree-width  $k$  has a nice tree decomposition of width  $k$ . Furthermore, if  $n$  is the number of vertices of  $G$  then there exists a nice tree decomposition with at most  $4n$  nodes. Moreover, this decomposition can be constructed in  $\mathcal{O}(n)$  time.*

In a nice tree decomposition  $(\mathcal{X}, T)$  every node is one of four possible types. These types are:

**Start** If a node is a leaf, it is called a *start node*.

**Join** If a node has two children, it is called a *join node* (note that the subgraphs of its children are then disjoint except for  $X_i$ ).

**Forget** If a node  $i$  has one child  $j$  and  $|X_i| < |X_j|$ , node  $i$  is called a *forget node*.

**Introduce** If a node  $i$  has one child  $j$  and  $|X_i| > |X_j|$ , node  $i$  is called an *introduce node*.

Moreover, we may assume that *start* nodes contain only a single vertex. If this is not the case, we can transform the nice tree decomposition into one having this property by adding a chain of *introduce* nodes in place of non-conforming *start* nodes. We will also need a notion of terminal graph, which is closely related to tree decompositions.

**Definition 4 (Terminal graph).** *A terminal graph is a triple  $H = (V, E, X)$ , where  $(V, E)$  is a graph and  $X$  an ordered subset of vertices of  $V$  called terminals. The operation  $\oplus$  is defined on pairs of terminal graphs with the same number of terminals:  $H \oplus H'$  is obtained by taking the disjoint union of  $H$  and  $H'$  and then identifying the  $i$ -th terminal of  $H$  with  $i$ -th terminal of  $H'$  for  $i \in \{1, \dots, l\}$ . A terminal graph  $H$  is a terminal subgraph of a graph  $G$  iff there exists a terminal graph  $H'$  s.t.  $G = H \oplus H'$ . Finally we define  $H_i$  to be  $D_i$  taken as a terminal subgraph with  $X_i$  as a set of its terminals (the ordering of  $X_i$  is not important here).*

## 4 The Algorithm for Parity Games

In this section we give the algorithm for solving parity games on graphs of bounded tree-width in polynomial time. First let us fix a parity game  $\mathcal{G} = (V_0, V_1, E, \lambda)$ . In the text we will often use only  $D = (V, E)$  (where  $V = V_0 \cup V_1$ ) to denote the game  $\mathcal{G}$  – sets  $V_0$ ,  $V_1$ , and  $\lambda$  are then implied by context. Then we take  $G$  to be the undirected graph which arises from  $D$  by forgetting the orientation of edges (i.e. an edge  $e = \{x, y\}$  of  $G$  can correspond to two edges  $(x, y)$  and  $(y, x)$  of  $D$ ). We also assume that we have a tree decomposition  $(\overline{\mathcal{X}}, \overline{T})$  of  $G$  of tree-width  $k$  and there is no cycle of length one. (If there is such a cycle, than the winner is known and we can safely remove the edge.) In the case of control flow graphs of programs, tree decomposition can be obtained by a simple syntactical analysis (see [17]).

We start by converting a tree decomposition  $(\overline{\mathcal{X}}, \overline{T})$  into a nice tree decomposition  $(\mathcal{X}, T)$  of the same width – this can be done using Lemma 1. Our algorithm then follows a general approach for solving problems on graphs of bounded tree-width (see [1]). The crux of the algorithm lies in computing a bounded rep-



resentation of the exponential set of strategies. Given node  $i$  of  $T$ , we only need to know effect of a given strategy for vertices in the interface  $X_i$ , size of which is bounded by a (small) constant. We compute the effects of strategies (called *full borders*) at nodes of  $T$  in a bottom-up manner. From a full border for the root we can then quickly decide the winner for vertices in the root node. Using force-sets or some similar technique, winners for the other vertices can be found as well (the complexity then increases by at most a factor of  $n$ ).

#### 4.1 Borders

In this section we will define the notion of *full border* for a graph  $G$  and show it is adequate. We need to have a means to record the “winner” for a set of paths.

To do that, we define the function  $\mathcal{R}$  (called *reward*) by the following prescription:

$$\mathcal{R}(p) = \begin{cases} -\lambda(v) & \text{iff } \lambda(v) \text{ is even} \\ \lambda(v) & \text{iff } \lambda(v) \text{ is odd} \end{cases}$$

We will overload  $\mathcal{R}$  so we can use it to talk about (finite) paths  $\mathcal{R}(\pi) = \max(\mathcal{R}(\pi_1), \dots, \mathcal{R}(\pi_n))$ . The intuition behind this definition is the following: In the game where there is a fixed strategy  $S$  for player  $P_0$  (i.e. every vertex in  $V_0$  has a single outgoing edge), it is the player  $P_1$  who decides which way to go if there are multiple choices available. To maximise her chances of winning, she chooses the path where the highest priority is odd, and the maximum of all such paths. If player  $P_0$  wins all the paths, then  $P_1$  tries to minimise the harm by selecting the one with the lowest winning priority. For a set of paths  $\Pi$  we define  $\overline{\mathcal{R}} = \max\{\mathcal{R}(\pi) \mid \pi \in \Pi\}$ . We will need also the following property, which is easy to prove:

**Lemma 2.** *Let  $D$  be a graph and  $u, v, w \in V$ . Let  $\Pi_D(u, v)$  be the set of all paths from  $u$  to  $v$  in  $D$ , and  $\Pi_D(u, v, w)$  set of those which pass through  $w$ . Then*

$$\overline{\mathcal{R}}(\Pi_D(u, v, w)) = \max(\overline{\mathcal{R}}(\Pi_D(u, w)), \overline{\mathcal{R}}(\Pi_D(w, v)))$$

A border of  $i$  tells us what happens *inside* the subgraph  $D_i$  – i.e. we take vertices of  $X_i$  as entry points for  $D_i$ , but not as its inner vertices. We start with some useful definitions.

An  *$i$ -path* in a graph  $D$  is path  $\pi : \pi_1 \pi_2 \dots \pi_k$ , where  $\pi_1 = s, \pi_k = t \in V_i$  and  $\{\pi_2 \dots \pi_{k-1}\} \subseteq V_i \setminus X_i$ . We allow  $s = t$ . For  $s, t \in X_i$  we use  $\Pi_D^i(s, t)$  to denote the set of all  $i$ -paths from  $s$  to  $t$  in  $D$ .

An  *$i$ -internal cycle* in a graph  $D$  is a cycle  $\sigma = \sigma_1 \sigma_2 \dots \sigma_k$ , where  $\sigma_1 = \sigma_k$  and  $\sigma_1, \dots, \sigma_k \in V_i \setminus X_i$ . Vertex  $w \in X_i$  is called  *$i$ -losing*, if there is an  $i$ -internal cycle  $\sigma$  won by  $P_1$  and a path  $\pi : \pi_1 \pi_2 \dots \pi_k$  s.t.  $\pi_1 = w, \pi_2 \dots \pi_k \in V_i \setminus X_i$  and  $\pi_k \in \sigma$ .

**Definition 5 (border).** A border  $b$  of  $i$  is a function  $b : X_i \rightarrow \{\perp, \circ, 2^{X_i \times P}\}$ . A border  $b$  (of  $i$ ) corresponds to a strategy  $S$ , iff  $\forall v \in X_i$ .  $b(v) =$

- $\perp$  iff  $v$  is  $i$ -losing in  $D^S$ ,
- $\circ$  iff  $v \in V_0$  has no outgoing edge in  $D_i^S$ , and
- $\{(w, p) \mid w \in X_i \wedge \Pi_{D^S}^i(v, w) \neq \emptyset \wedge p = \overline{\mathcal{R}}(\Pi_{D^S}^i(v, w))\}$  otherwise.

First note that many strategies can correspond to a single border. This “compression” makes the algorithm work. Members of  $b(s)$  are called *entries*. Note that for each  $v, w \in X_i$  there is at most one entry  $(w, p) \in b(v)$  when  $b(v) \neq \perp$  or  $\circ$ . For the sake of clarity, we will overload the notation a little bit and write  $b(s, t) = p$  as a shorthand for  $(t, p) \in b(s)$  and  $b(s, t) = 0$  when there is no pair  $(t, p) \in b(s)$ . In addition to border being a function, we can look at it as being a table of priorities with dimensions  $(k + 1) \times (k + 1)$ . In this table the rows and columns are labelled by vertices. Likewise if symbols  $\perp$  or  $\circ$  appear in a row, then the whole row must be marked by this symbol.

Next we want to show that border (corresponding to  $S$ ) is well defined – i.e. that it contains all the information we need to know about  $D_i^S$  in order to check whether  $P_0$  wins using the strategy  $S$ . To do this we first define a notion of *link*:

**Definition 6 (link).** *A link of a border  $b$  (of a node  $i$ ) is a terminal graph  $H = (X_i \cup W \cup \{v_\perp\}, E \cup (v_\perp, v_\perp), X_i)$  and priority function  $\lambda$  s.t.:*

- *for every pair  $s, t \in X_i$  with  $b(s, t) \neq 0$  we put a new vertex  $w$  into  $W$  and two edges  $(s, w)$  and  $(w, t)$  into  $E$ . We also set  $\lambda(w) = b(s, t)$ .*
- *for every  $s \in X_i$  with  $b(s) = \perp$  we insert an edge  $(s, v_\perp)$  into  $E$ .*
- *$p(v)$  for  $v \in X_i$  is the same as in original game,  $\lambda(v_\perp) = 1$ .*

*We also write  $Link(b)$  for the link of the border  $b$ .*

In other words a link of  $b$ , where  $b \in B(i)$  corresponds to a strategy  $S$ , is just a graph having the same properties w.r.t. winning the game as  $D_i^S$  does. The formal proof of this statement is subject of the following lemma:

**Lemma 3 (Border is well defined).** *Let  $b$  be a border of  $i$  corresponding to some strategy  $S$ . Let  $H$  be s.t.  $D^S = D_i^S \oplus H$  and  $v$  a vertex in  $V \setminus (V_i \setminus X_i)$ . Then  $P_1$  has a winning strategy for  $v$  in  $D^S$  iff she has a winning strategy for  $v$  in  $L = Link(b) \oplus H$ .*

*Proof.*  $\Rightarrow$  Suppose  $P_0$  does not win in  $D^S$ . Then there must be a cycle  $\sigma = \sigma_1 \dots \sigma_k$  reachable from  $v$  s.t.  $P_1$  wins this cycle. Let  $\pi$  be the path from  $v$  to a vertex of  $\sigma$ . There are two cases to be considered:

1.  $V(\sigma) \subseteq V_i \setminus X_i$

Then there must be  $i \in \mathbb{N}$  s.t.  $\pi_i = w \in X_i$  and  $b(w) = \perp$  by Fact 1. From definition of  $Link(b)$  player  $P_1$  has a winning strategy for  $v$  in  $L$  (she can force play to  $v_\perp$  and then loop through this vertex).

2. Otherwise

Let  $j$  be s.t.  $\sigma_j \in X_i$  and  $\forall i \leq j. \sigma_i \in V_i \setminus X_i$  (such a  $j$  must exist). Then  $\sigma_j$  is also reachable in  $L$  (by definition of border correspondence and  $Link(b)$ ). Moreover, let  $\sigma'$  be a cycle obtained from  $\sigma$  by substituting every sequence  $s = \sigma_i \dots \sigma_{i+l} = t$ , where  $s, t \in X_i$  and  $\{\sigma_{i+1} \dots \sigma_{i+l}\} \subseteq V_i \setminus X_i$ , by path  $sw_{s,t}t$ . Then  $\sigma'$  is a cycle of  $L$  and it is easy to check that  $P_1$  also wins the cycle  $\sigma'$  of  $L$ .

$\Leftarrow$  similar

□

**Definition 7 (full border).** A full border  $B(i)$  (of node  $i$ ) is just a set of all borders of  $i$  corresponding to some strategy  $S$ .

The following important corollary says how we can derive the desired information from the full border for the root  $r$  of  $T$ .

**Corollary 1.** Let  $(\mathcal{X}, T)$  be a tree decomposition of  $D$ ,  $r$  its root node and  $v \in X_r$  a vertex of  $D$ . Then  $P_0$  has a winning strategy for  $v$  in  $D$  iff there is  $b \in B(r)$  s.t.  $P_0$  has a winning strategy for  $v$  in  $\text{Link}(b)$ .

It should be noted that the check in the corollary above can be done in constant time, which depends only on the tree-width of  $D$ .

## 4.2 Computing Full Border

Having a nice tree decomposition  $(\mathcal{X}, T)$ , we compute  $B(i)$  for every node  $i$  of  $T$  in a bottom-up manner. Here we give an algorithm for each of the four node types. Detailed algorithms can be found in the full version of the paper [13].

**Start Node** If  $i$  is a Start node, then it contains only a single vertex  $v$ . Two cases:

- $v \in V_0$  - we set  $B(i) = \{b\}$  where  $b(v) = \circ$ , since we have to postpone the choice.
- $v \in V_1$  - we set  $B(i) = \{b\}$  where  $b(v) = \emptyset$  (no choice here).

**Forget Node** – *detecting cycles*. Let  $i$  be a forget node with a single child  $j$  and  $X_j = X_i \cup \{v\}$ . By definition of tree-width we know that there is no edge connecting  $v$  with  $D \setminus V_i$ , since  $v$  does not appear anywhere in the part of  $T$  yet to be explored. For  $b \in B(j)$ :

1. Create  $b'$  from  $b$  by copying all entries not containing  $v$ .
2. If  $b(v) = \perp$  or  $b(v, v) = p$  for  $p$  odd, set  $b'(u) = \perp$  for all vertices  $u \in X_i$  s.t.  $b(u, v) \neq 0$ , since  $u$  is  $i$ -losing.
3. Otherwise let  $s, t$  be a pair of vertices s.t.  $b(s, v) = p_1$  and  $b(v, t) = p_2$  and let  $p = \max(p_1, p_2)$ . Then  $b'(s, t) = p$  if  $b(s, t) = 0$ , and  $b'(s, t) = \overline{\mathcal{R}}(b(s, t), p)$  otherwise.
4. Put  $b'$  into  $B(i)$ .

**Introduce Node** – *adding new borders*. Let  $i$  be an introduce node with a single child  $j$  and  $X_i = X_j \cup \{v\}$ . For every border  $b \in B(j)$ :

1. Create a copy  $b'$  of  $b$ .
2. For every edge  $(v, w) \in E$  s.t.  $w \in X_i$  insert an entry  $b'(v, w) = \max(\lambda(v), \lambda(w))$ . If  $v \in V_1$ , then we add all such entries. Otherwise  $v \in V_0$  and we create a new border for each of the edges (these borders correspond to different strategies) and also include the possibility  $b'(v) = \circ$ .

3. For every border  $b'$  created in the previous step and every edge  $(u, v) \in E, u \in X_i \cap V_1$  we set  $b'(u, v) = \max(\lambda(u), \lambda(v))$ .
4. For every  $W \subseteq \{u \in X_i \mid (u, v) \in E \wedge b(u) = \circ\}$  we create a new copy  $b''$  of  $b'$  with  $b''(u, v) = \max(\lambda(u), \lambda(v))$  for  $u \in W$ . Every such subset may correspond to some strategy  $S$  for  $D$ .
5. Remove every newly created  $b$  which can not correspond to a strategy. This happens when there is  $v' \in X_i$  s.t.  $b(v') = \circ$ , but there is no  $w \in V \setminus V_i$  s.t.  $(v', w) \in E$ . In other words - we have postponed the choice, but there are no remaining edges to choose from.
6. Add the resulting borders to  $B(i)$ .

**Join Node** Let  $i$  be a join node with  $j_1$  and  $j_2$  as its children and  $B(j_1)$  and  $B(j_2)$  their full borders. It is enough to define the join operation for every pair  $(b_1, b_2)$  where  $b_1 \in B(j_1)$  and  $b_2 \in B(j_2)$ . Each such join will result in a new border  $b$  to be added into full border  $B(i)$  of  $i$ .

*Note on joining* To get correct results, we must not apply the join operation to two borders which do not correspond to some common strategy. By definition of tree-width  $V_{j_1} \cap V_{j_2} = X_i$  (i.e.  $D_{j_1}$  and  $D_{j_2}$  are disjoint except for their common interface), so we only have to check the choices made for  $P_0$  vertices in  $X_i$ . To allow for the check, in every border we remember the choice made for these vertices. Look at the following table. For every  $v \in X_i \cap V_0$  there is a single edge  $(v, w)$  in  $D^S$ . The possible cases are:

- |                                  |                                  |
|----------------------------------|----------------------------------|
| 1) $w \in V \setminus V_i$       | 2) $w \in X_i$                   |
| 3) $w \in V_{j_1} \setminus X_i$ | 4) $w \in V_{j_2} \setminus X_i$ |

To distinguish among these cases, we already can deduce some information from  $b_1(v)$  and  $b_2(v)$  (we use  $S, S'$  to mark the fact that  $b(s)$  is a non-empty set of pairs):

$b_1(v) \setminus b_2(v)$	$\circ$	$\perp$	$S'$
$\circ$	1)	4)	4?
$\perp$	3)	—	—
$S$	3?	—	2?

As we see, there are some cases which need further checking (these are marked “?” in the table). Note that  $b(v, w) \neq 0$  when  $w$  is in  $X_i$ , so we can remember the choice by selecting a single pair from  $b(v)$  for every  $v \in X_i \cap V_0$ . No pair selected then corresponds to  $w$  being in  $V_i \setminus X_i$ . Our algorithms can be easily extended to keep track of this information.

*Algorithm* For a pair of borders  $b_1$  and  $b_2$  we first check whether they can be both borders of a same strategy as outlined above. Now we have to “merge”  $b_1$  and  $b_2$  into  $b$  by going through all vertices in  $X_i$ . The only interesting case is when  $b_1(s, t) = p_1$  and  $b_2(s, t) = p_2$ , in which case we set  $b(s, t) = \overline{\mathcal{R}}(p_1, p_2)$ . Once finished, we must remove borders which cannot correspond to a strategy (see the Introduce node).

### 4.3 Correctness and Complexity

The following theorem states that our algorithm is correct.

**Theorem 2 (correctness).** *Let  $\mathcal{G} = (V_0, V_1, E, \lambda)$  be a parity game and  $(\mathcal{X}, T)$  a nice tree decomposition of  $D$ . Let  $i$  be a node of  $T$  and  $B(i)$  the set computed using our algorithm for a node  $i$ . Then  $B(i)$  is a full border for  $i$ .*

*Proof (Sketch).* The proof goes by induction on structure of  $T$ . For every node we have to prove that

1. For every strategy  $S$  there is  $b \in B(i)$  corresponding to  $S$
2. Every  $b \in B(i)$  corresponds to some strategy  $S$ .

The proof itself is straightforward, since all the necessary facts were mentioned in Sect. 4.2 alongside the algorithms.  $\square$

**Theorem 3 (complexity).** *Let  $\mathcal{G} = (V_0, V_1, E, \lambda)$  be a parity game with tree decomposition  $(\mathcal{X}, T)$  of  $D = (V = V_0 \cup V_1, E)$  of width  $k$ . Let  $\mathbf{p} = |\{\lambda(v) \mid v \in V\}|$  be the number of priorities. The algorithm described in this section runs in time  $\mathcal{O}(n \cdot k^2 \mathbf{p}^{2(k+1)^2})$  where  $n$  is the number of vertices of  $D$ .*

*Proof.* There are at most  $\mathbf{p}^{(k+1)^2}$  different borders in a full border. That is because the dimensions of a single border are at most  $(k+1) \times (k+1)$ , and a border is nothing else than a table of priorities. It can be easily seen that full border for each of the four node types can be computed in constant time depending only on  $k$ . A precise analysis shows that this time has an upper bound of  $k^2 \cdot \mathbf{p}^{2(k+1)^2}$ .

According to Lemma 1 we know that a nice tree decomposition has at most  $4n$  nodes and can be constructed in  $\mathcal{O}(n)$  time. This gives the bound  $\mathcal{O}(n \cdot k^2 \mathbf{p}^{2(k+1)^2})$ . It remains to mention that in the general case the number of priorities  $\mathbf{p}$  is from the range  $\langle 1, n \rangle$ , and therefore our algorithm is *polynomial* in  $n$ .  $\square$

We have been able to identify examples of parity games for which the standard algorithm based on computing the approximants needs exponential time, but which are of very low tree-width. In [12] there is an example of such a parity game. This example is parametrized by  $n$  – the number of vertices. The game graph in Fig. 1 shows an instance of size 10. In our notation vertices of  $P_0(P_1)$  are shown as circles (squares) and the number associated with each vertex shows its priority. Note that the tree-width of this game graph is only 2 (this value does not depend on  $n$ ).

## 5 Adaptation to $\mu$ -Calculus

In this section we explain how to adapt the algorithm for parity games to  $\mu$ -calculus model checking. As an instance of a model checking problem, we are

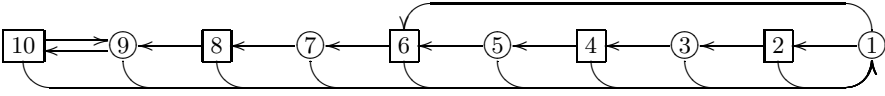


Fig. 1. Parity game example

given an LTS  $\mathcal{T}$  of size  $n$  and a  $\mu$ -calculus formula  $\varphi$  of size  $m$ . Moreover, we assume that  $\mathcal{T}$  has a tree decomposition of tree-width  $k$  and therefore also a nice tree decomposition  $(\mathcal{Y}, T)$ , where  $T = (I, F)$ ,  $\mathcal{Y} = \{Y_i \mid i \in I\}$ , of the same size. For  $D$  we take the corresponding game graph created using the translation from Sect. 2.2.

It is important to realize that a direct approach by taking a parity game graph  $D$  and using the previous algorithm to solve  $D$  does not work as intended. This is because of the fact that  $D$  can be of much higher tree-width than  $\mathcal{T}$ , because the graph of the formula  $\varphi$  can contain several loops. Actually, the following fact holds:

**Fact 2.** *The graph of a  $\mu$ -formula  $\varphi$  is of tree-width at most  $n$ , where  $n$  is the number of variables in the formula (and not any greater than alternation depth). Moreover for every  $n$  there exists a formula  $\varphi$  with  $n$  variables such that the tree-width of the graph of  $\varphi$  is  $n$ .*

Instead of computing the game graph first, we can just use the algorithm for parity games on the graph of the system being checked. The only change we make is that instead of adding/removing a single vertex  $p$ , we will add/remove  $m$  vertices  $(p, \psi)$  at a time – one for every  $\psi \in \text{Sub}(\varphi)$  ( $m = |\text{Sub}(\varphi)|$ ).

Taking  $(\mathcal{Y}, T)$  and  $D$  as above, we define  $X_i = \{(p, \psi) \mid p \in Y_i, \psi \in \text{Sub}(\varphi)\}$ . It is easy to check that using these  $X_i$ 's and  $D$  in the definition of border makes all the results in Sect. 4.1 hold. Now we must modify the algorithm a little bit. We will progress on the structure of  $\mathcal{Y}$ :

*Introduce node* A vertex  $v$  of  $T$  is being introduced. We will add vertices  $(v, \psi)$  one by one, as a sequence of vertices in the original algorithm. The order is not important here.

*Start node* A leaf containing only  $v$  is created. In this case we create a border with vertex  $(v, \varphi)$  and then add vertices  $(v, \psi)$ ,  $\psi \in \text{Sub}(\varphi)$  one by one, as in the original introduce node algorithm.

*Forget node* Vertex  $v$  of  $T$  is being removed. We remove the vertices  $(v, \psi)$  one by one as a sequence of vertices in the original algorithm.

*Join node* The only different bit is checking whether two borders can correspond to some common strategy. The choices made in the vertices of type  $(v, \langle a \rangle \psi)$  are checked by the original algorithm, as the edge goes to some vertex  $(w, \psi)$ , where  $v \neq w$ . The choices made for vertices  $(v, \psi_1 \vee \psi_2)$  can be checked easily, as there is an edge to either  $(v, \psi_1)$  or  $(v, \psi_2)$  in both the borders.

## 5.1 Complexity

**Theorem 4 (complexity).** *Let  $\mathcal{T}$  be a LTS of size  $n$  with a tree decomposition of tree-width  $k$ , and  $\varphi$  a formula of size  $m$ . Then the previous algorithm runs in time  $\mathcal{O}(n \cdot (km)^2 d^{2((k+1)m)^2})$ .*

*Proof.* We start with the complexity estimate for parity games. In the  $\mu$ -calculus case, the size of borders has grown from  $k + 1$  to  $(k + 1) \cdot m$ . However, we do not increase the number of nodes in tree-decomposition. The number of priorities  $\mathbf{p}$  is bounded by  $m$  (actually, we can bound it by  $d$ , the alternation depth of formula). The rest follows from Theorem 3.  $\square$

Comparing to the result of [11], our algorithm is *linear* in the size of the system, no matter what the formula is. It should be also noted, that the estimated running time is really the upper bound and the algorithm may benefit from further optimisation.

## 5.2 Application to Software Model Checking

The algorithm presented above looks suitable for model checking software programs. For structured programs have a low tree-width and, moreover, we can find the tree decomposition just by performing a simple syntactic analysis [17]. In practice it is usually the case that the size of the system itself is huge, whereas the formula is quite small. This is where the fact that our algorithm is linear in the size of the system may give better results compared to previous algorithms.

## 6 Conclusions and Future Work

We have shown that parity games can be solved in polynomial time for the important class of systems of bounded tree-width. This result was then used to present  $\mu$ -calculus model checking algorithm which is linear in the size of the system. We hope that these results can bring another insight into what is the exact complexity of solving parity games and  $\mu$ -calculus model checking. In addition software model checking may benefit from this work, since control flow graphs of structured programs have bounded tree-width.

Following the approach presented here, there is a hope to obtain even better results. For example, our algorithm is not optimal on directed acyclic graphs (DAGs). Even though  $\mu$ -calculus model checking (or solving parity games) on DAGs can be done in linear time, DAGs can have a high tree-width. This is because we decompose the underlying undirected graph, and therefore do not take into account some knowledge we have about the system. Finding some right structural property of directed graphs might prove useful.

## References

1. Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *MFCS'97*, volume 1295 of *LNCS*, pages 19–36, 1997.
2. B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.
3. E. Emerson and C. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 5th IEEE Foundations of Computer Science*, pages 368–377, 1991.
4. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *CAV 93*, volume 697 of *LNCS*, pages 385–396. Springer-Verlag, 1993.
5. E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional  $\mu$ -calculus. In *Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, June 1986.
6. M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS'02*, pages 215–224. IEEE Computer Society, 2002.
7. J. Gustedt, O. Mæhle, and J. A. Telle. The treewidth of Java programs. In *Proceedings of ALENEX'02*, volume 2409 of *LNCS*. Springer-Verlag, 2002.
8. M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, volume 1770 of *LNCS*, pages 290–301. Springer-Verlag, 2000.
9. T. Kloks. *Treewidth – computations and approximations*, volume 842 of *LNCS*. Springer-Verlag, 1994.
10. D. Kozen. Results on the propositional  $\mu$ -calculus. *Theoretical Computer Science (TCS)*, 27:333–354, 1983.
11. D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV '94*, volume 818 of *LNCS*, pages 338–350. Springer-Verlag, 1994.
12. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8, Bertz Verlag, Berlin, 1997.
13. J. Obdržálek. Fast mu-calculus model checking when tree-width is bounded. Technical report, LFCS, July 2003. <http://www.dcs.ed.ac.uk/home/s0128832>.
14. N. Robertson and P. D. Seymour. Graph Minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49–63, 1984.
15. C. Stirling. Local model checking games. In *CONCUR '95*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.
16. Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, 2001.
17. M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.
18. J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV 2000*, volume 1855 of *LNCS*, pages 202–215. Springer-Verlag, 2000.



# Dense Counter Machines and Verification Problems

Gaoyan Xie<sup>1</sup>, Zhe Dang<sup>1\*</sup>, Oscar H. Ibarra<sup>2\*\*</sup>, and Pierluigi San Pietro<sup>3\*\*\*</sup>

<sup>1</sup> School of Electrical Engineering and Computer Science  
Washington State University  
Pullman, WA 99164, USA

<sup>2</sup> Department of Computer Science  
University of California  
Santa Barbara, CA 93106, USA

<sup>3</sup> Dipartimento di Elettronica e Informazione  
Politecnico di Milano, Italia

**Abstract.** We generalize the traditional definition of a multcounter machine (where the counters, which can only assume nonnegative integer values, can be incremented/decremented by 1 and tested for zero) by allowing the machine the additional ability to increment/decrement each counter  $C_i$  by a nondeterministically chosen fractional amount  $\delta_i$  between 0 and 1 ( $\delta_i$  may be different at each step). Further at each step, the  $\delta_i$ 's of some counters can be linearly related in that they can be integral multiples of the same fractional  $\delta$  (e.g.,  $\delta_1 = 3\delta$ ,  $\delta_3 = 6\delta$ ). We show that, under some restrictions on counter behavior, the binary reachability set of such a machine is definable in the additive theory of the reals and integers. There are applications of this result in verification, and we give an example in the paper. We also extend the notion of “semilinear language” to “dense semilinear language” and show its connection to a restricted class of dense multcounter automata.

## 1 Introduction

Dense counters are necessary in modeling many control systems and real-time applications. Their importance has already been noticed in model-checking. One of the focuses during the past ten years is the study of hybrid automata [2,13] where dense counters follow some complex continuous flow, typically characterized by differential equations, and interact with bounded discrete control variables. Decidable results on reachability has been obtained for many forms of hybrid automata (e.g., timed automata [3], multirate automata [2,18], initialized rectangular automata [19]). On the other hand, linear hybrid automata in general are undecidable for reachability [19]. In fact, the undecidability remains even for timed automata augmented with one stop watch [19].

Another focus approaches dense counters in a different way. Ultimately, in the view of an external observer, the evolution of a system containing dense counters in  $\mathbf{X}$  can

---

\* Corresponding author (zdang@eecs.wsu.edu).

\*\* The research of Oscar H. Ibarra has been supported in part by NSF Grants IIS-0101134 and CCR02-08595.

\*\*\* Supported in part by MIUR grants FIRB RBAU01MCAC, COFIN 2001015271, CNR Strategico 1999-Società dell'Informazione -SP4.

be characterized by a formula  $T(s, \mathbf{X}, s', \mathbf{X}')$ , whose meaning is roughly: the system transits from control state  $s$  to  $s'$  while changing the counters from  $\mathbf{X}$  to  $\mathbf{X}'$ . In other words,  $T$  characterizes a one-step transition of the system, called a dense counter system. A fundamental question is then: what kinds of  $T$  would make the transitive closure  $T^*$  computable? Many model-checking queries (e.g., reachability) relies on the answer. Some results studying special forms of  $T$  and its (restricted) transitive closure can be found in e.g., [6,7,20,5,12].

Recently, it has been shown that some fundamental results in automata theory (such as [14,15,8]) on some restricted discrete counter machines are quite useful in studying various model-checking problems for infinite state systems containing discrete counters (e.g., [9,11] etc.). Following this line, we believe that developing a fundamental theory (which does not exist) in the view of automata theory for these machines but with dense counters should be equally useful for studying dense counter systems. This automata-theoretic approach is different from those taken in [6,20,5,12] mentioned earlier. As a starting point, in this paper we will develop a preliminary automata theory for dense counter machines.

We define a dense counter machine  $\mathcal{M}$  as a finite state machine augmented with a number of dense counters. The counters can assume nonnegative real values. At each step, each counter  $C_i$  can be incremented/decremented by 0, 1 or a nondeterministically chosen fractional amount  $\delta_i$  between 0 and 1 ( $\delta_i$  may be different at each step). We will see that without loss of generality, we can assume that at a single step, for all  $i$ , the  $\delta_i$ 's are of the same amount  $\delta$  (but still the  $\delta$  may be different between steps). This  $\delta$ -increment/decrement is the essential difference between dense and discrete counters. The machine  $\mathcal{M}$  can also test a counter against 0 ( $= 0?$ ,  $> 0?$ ). Since counters are assumed nonnegative,  $\mathcal{M}$  crashes if a counter falls below 0. (Note that since the counters can be tested against 0, the system can actually check if it will crash and if so enter a distinguished state. So the assumption of “crashing” can be made w.l.o.g.) Given two designated states  $s_{\text{init}}$  and  $s_{\text{final}}$  in  $\mathcal{M}$ , we study the possibility of computing the transitive closure “ $\leadsto$ ”, called the binary reachability, of  $\mathcal{M}$ :

$\mathbf{X} \leadsto \mathbf{X}'$  iff  $\mathbf{X}$  at  $s_{\text{init}}$  reaches  $\mathbf{X}'$  at  $s_{\text{final}}$  in  $\mathcal{M}$ .

On the negative side, even the state reachability problem (whether  $s_{\text{init}}$  reaches  $s_{\text{final}}$ , or equivalently, whether  $\leadsto$  is empty) is in general undecidable. This is because a two (discrete) counter machine can be simulated by  $\mathcal{M}$  in which no  $\delta$ -increments/decrements are made. But, what if  $\mathcal{M}$  only performs  $\delta$ -increments/decrements (along with tests against 0)? In this case, the undecidability remains even when there are only four dense counters. Interestingly, still in this case, the state reachability problem becomes decidable when there are two counters. The case for three counters is open.

On the positive side, we will show some restricted versions of  $\mathcal{M}$  whose binary reachability is definable in the additive theory of the reals and integers. The theory is decidable, for instance, by the Büchi-automata based decision procedure presented in [4] and a quantifier elimination technique in [21]. This will allow us to automatically verify some safety properties for the restricted  $\mathcal{M}$ : Are there  $\mathbf{X}$  and  $\mathbf{X}'$  such that  $I(\mathbf{X}, \mathbf{X}') \wedge \mathbf{X} \leadsto \mathbf{X}'$ ? where  $I$  is definable in the additive theory of the reals and integers (e.g.,  $x_1 + x'_1 - 3x_2 > 4 \wedge x_3 - x'_2 \text{ is an integer} \wedge x'_2 - x_1 < 7$ ). Furthermore, some liveness properties (e.g., infinitely oftenness [11]) can also be automatically verified:

Are there  $\mathbf{X}^0, \dots, \mathbf{X}^n, \dots$  such that  $\mathbf{X}^0 \rightsquigarrow \mathbf{X}^1 \rightsquigarrow \dots \rightsquigarrow \mathbf{X}^n \rightsquigarrow \dots$ ,  $I(\mathbf{X}_0)$  holds, and, there are infinitely many  $n$  for which  $P(\mathbf{X}_n)$  holds?

where  $I$  and  $P$  are definable in the additive theory of the reals and integers.

In this paper, we first consider a restricted version of  $\mathcal{M}$  in which all the dense counters are monotonic (i.e., never decreasing). Then we consider the case when all the dense counters are reversal-bounded (the alternations between nondecreasing and nonincreasing are bounded by a fixed integer, for each counter). The concept of “reversal-boundedness” is borrowed from its counterpart for discrete counters [14]. It is not too difficult to prove that in these cases  $\rightsquigarrow$  is definable in the additive theory of the reals and integers. The proof becomes more difficult when  $\mathcal{M}$  is further augmented with an unrestricted dense counter (called a free counter). In the rather complex proof, we use a “shifting” technique to simplify the behavior of the free counter.

The proofs can be easily generalized to the cases when counters in  $\mathcal{M}$  are of multirate. That is, a nonnegative integer rate vector  $(r_1, \dots, r_k)$  is now associated with a counter instruction such that, for each counter  $x_i$  ( $1 \leq i \leq k$ ), an increment/decrement by 1 now corresponds to an increment/decrement by  $r_i$ ; an increment/decrement by  $\delta$  now corresponds to an increment/decrement by  $r_i \cdot \delta$ . Also note that, in an instruction, the  $\delta$  need not necessarily be the same for each counter. For instance, one may have an instruction by associating  $\delta_1$  with  $x_1, x_2$  and  $\delta_2$  with  $x_4, x_5$  as follows:

```
s: if  $x_1 > 0$  then for some  $0 < \delta_1, \delta_2 < 1$ ,
       $x'_1 = x_1 + 3 \cdot \delta_1$ ,  $x'_2 = x_2 - 4 \cdot \delta_1$ ,  $x'_3 = x_3 + 5 \cdot 1$ ,
       $x'_4 = x_4 - 6 \cdot \delta_2$ ,  $x'_5 = x_5 - 7 \cdot \delta_2$ ,
goto  $s'$ .
```

The automata-theoretic approach taken in this paper is new and the decidable results for the transitive closure computations on the counter systems are incomparable to previously known results (e.g., [6,20,5,12]). The models, e.g., the reversal-bounded dense counter machines with a free counter, are incomparable with some restricted and decidable models for hybrid systems (e.g., timed automata [3], initialized rectangle automata [19], etc). In the future, we plan to use the results in this paper as a fundamental tool to reason about a broader class of dense counter systems. For instance, we may study a subclass of timed automata whose accumulated delays [1] are decidable over a formula of the additive theory of reals and integers. Our models are convenient in modeling accumulated delays which are simply monotonic dense counters. We can also use the model to specify some interesting systems with multiple dense/discrete counters.

Consider, e.g., the following version of the traditional producer-consumer system. A system may *produce*, *consume* or be *idle*. When in state *produce*, a resource is produced, which may be stored and later consumed while in state *consume*. The system may alternate between production and consumption states, but it may not produce and consume at the same time. Production may be stored, and used up much later (or not used at all). The novelty is that the resource may be a real number, representing for instance the available amount of a physical quantity (e.g., fuel, water or time).

This system may be easily modeled by a finite state machine with a free purely-dense counter, where the resource is added when produced or subtracted when consumed. Since the free counter may never decrease below zero (e.g., it can be tested against 0), the system clearly implements the physical constraint that consumption must never

exceed production. Using a dense-counter, there is an underlying assumption that a continuous variable, such as our resource, changes in discrete steps only; however, this is acceptable in many cases where a variable actually changes continuously, since the increments/decrements may be arbitrarily small.

We note that decidable versions of timed or hybrid automata do not appear to be able to specify the producer-consumer example above (even though we cannot rule out that some other decidable model can). In timed automata, when the resource is interpreted as time, the amount of production (respectively, consumption) is equivalent to the “total accumulated time spent while in state *produce* (respectively, *consume*)”. Implementing the constraint  $production - consumption \geq 0$  is then tantamount to implementing a stop watch, which is known to lead to undecidability of the model. In hybrid automata, a continuous variable for  $production - consumption$  must have different flow rates in state *produce* (nonnegative) and in state *consume* (nonpositive). A rate change, at least for the decidable class of initialized rectangular automata, forces the variable to be reinitialized to a predefined value after a state change: the value a variable had in state *produce* is lost in state *consume*. Adding more variables does not appear to solve the problem, either, since no variable comparison is possible.

The dense-counter model of the example can be easily adapted to more complex situations. For instance, suppose that a constraint on the system is that total production is at most twice the amount of consumption. This can be described by adding two reversal-bounded dense counters, namely  $p$ ,  $c$ , to store the total production ( $p$ ) and twice the total consumption ( $c$ ). For the sake of clarity, a nondeterministic version of the producer-consumer model  $\mathcal{M}$  is shown in Figure 1, in which counter  $count$  denotes the difference between the total production and the total consumption. Each arc has a boolean guard and may specify an increment or decrement that is a multiple of a  $\delta$ ,  $0 < \delta < 1$ , for each counter  $count$ ,  $p$ ,  $c$ . Recall that  $\delta$  may be different at each step. If no variation is specified, then the counter stays. When in the (initial) state *idle*,  $\mathcal{M}$  may

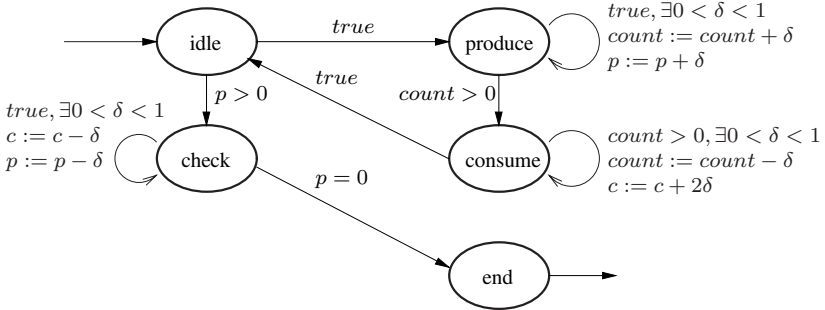


Fig. 1. A producer-consumer system  $\mathcal{M}$ .

start producing or, nondeterministically, may go into the checking state if  $p > 0$ . While producing (in state *produce*), counter  $p$  is increased together with  $count$  (hence, of the same amount). Eventually consumption starts: counter  $c$  is increased in state *consume* of

the double amount used to decrease *count*. Then  $\mathcal{M}$  may go back to the *idle* state. Notice that *count* may never go below 0, otherwise  $\mathcal{M}$  crashes (alternatively, *count* could be tested against 0, but crashing is ruled out since it leads anyway to a nonaccepting path).

When finished producing and consuming,  $\mathcal{M}$  goes into state *check*, where it decrements both *p* and *c* until *p* goes to 0. At this time, it moves to the final state *end*. Hence, if the value of *c* is not greater or equal to *p*,  $\mathcal{M}$  crashes. Nondeterminism allows the automaton to guess a sequence of decrements  $\delta$  that leads to *p* = 0 and hence to the final state *end*. To verify that the system really produces at most twice than it consumes, it is enough to check whether  $\mathcal{M}$  may reach state *end*. Similar linear bounds, such as “total production should not exceed consumption more than 5 per cent”, can be dealt with analogously.

A more sophisticated verification capability is given by the fact that the binary reachability  $\leadsto^{\mathcal{M}}$  is not only computable but may be represented with a formula in the additive theory of reals and integers (hence, decidable). For instance, consider the property that the system may be implemented with finite storage (i.e., *count* is bounded). This can be expressed by the following decidable formula:

$$\exists y > 0 \forall s \forall \mathbf{X} ((idle, \mathbf{0}) \leadsto^{\mathcal{M}} (s, \mathbf{X}) \Rightarrow \mathbf{X}(count) \leq y)$$

whose meaning is that there is a bound  $y > 0$  such that, starting in *idle* state with all counters equal to 0,  $\mathcal{M}$  can only reach configurations  $(s, \mathbf{X})$  where the value of *count* in  $\mathbf{X}$ ,  $\mathbf{X}(count)$ , is not greater than  $y$ .

In the simple case of Figure 1, the property is obviously violated since for instance production in state *produce* may go on unbounded without consumption.

Due to space limitation, proofs are omitted in the paper. The full version of the paper is accessible at [www.eecs.wsu.edu/~zhang](http://www.eecs.wsu.edu/~zhang).

## 2 Preliminaries

A dense counter is a nonnegative real variable. A dense multicounter machine  $\mathcal{M}$  is a finite state machine augmented with a number of dense counters. On a move from one state to another,  $\mathcal{M}$  can add an amount of 0, +1, -1, + $\delta$ , or - $\delta$  for some nondeterministically chosen  $\delta$ , with  $0 < \delta < 1$ . On a move,  $\mathcal{M}$  can also test a counter against 0 while leaving all the counters unchanged. Formally, we have the following definitions. A dense counter changes according to one of the following five *modes*: stay, unit increment, unit decrement, fractional increment, fractional decrement. We use  $\mathbf{X}$  to denote a vector of values for dense counters  $x_1, \dots, x_k$ . In the sequel, we shall use  $\mathbf{X}(x_i)$  to denote the component for  $x_i$  in  $\mathbf{X}$ . A *mode vector* is a  $k$ -tuple of modes. There are  $5^k$  different mode vectors. Given a mode vector  $\mathbf{m} = (m_1, \dots, m_k)$  and an amount  $0 < \delta < 1$ , we define

$$R_{\mathbf{m}, \delta}(\mathbf{X}, \mathbf{X}')$$

as follows:  $R_{\mathbf{m}, \delta}(\mathbf{X}, \mathbf{X}')$  iff for each  $1 \leq i \leq k$ ,  $\mathbf{X}(x_i) \geq 0$  and  $\mathbf{X}'(x_i) \geq 0$ , and each of the followings holds:

- if  $m_i$  is stay, then  $\mathbf{X}'(x_i) = \mathbf{X}(x_i)$ ;
- if  $m_i$  is unit increment, then  $\mathbf{X}'(x_i) = \mathbf{X}(x_i) + 1$ ;

- if  $m_i$  is unit decrement, then  $\mathbf{X}'(x_i) = \mathbf{X}(x_i) - 1$ ;
- if  $m_i$  is fractional increment, then  $\mathbf{X}'(x_i) = \mathbf{X}(x_i) + \delta$ ;
- if  $m_i$  is fractional decrement, then  $\mathbf{X}'(x_i) = \mathbf{X}(x_i) - \delta$ .

We use

$$R_{\mathbf{m}}(\mathbf{X}, \mathbf{X}')$$

to characterize the relationship between the old values  $\mathbf{X}$  and the new values  $\mathbf{X}'$  under the mode vector; i.e.,  $R_{\mathbf{m}}(\mathbf{X}, \mathbf{X}')$  is true iff there exists some  $0 < \delta < 1$  such that  $R_{\mathbf{m}, \delta}(\mathbf{X}, \mathbf{X}')$  holds. Notice that when  $\mathbf{m}$  does not contain any mode of fractional increment/decrement, the amount  $\delta$  is irrelevant.

Formally, a (*nondeterministic*) *dense counter machine*  $\mathcal{M}$  is tuple

$$\langle S, \{x_1, \dots, x_k\}, s_{\text{init}}, s_{\text{final}}, T \rangle, \quad (1)$$

where  $S$  is a finite set of (*control*) *states* where  $s_{\text{init}}$  and  $s_{\text{final}}$  are the initial and the final states.  $x_1, \dots, x_k$  are dense counters.  $T$  is a set of transitions. Each transition is a triple  $(s, \mathbf{m}, s')$  of a source state  $s$ , a mode vector  $\mathbf{m}$ , and a destination state  $s'$ . A configuration of  $\mathcal{M}$  is a pair  $(s, \mathbf{X})$  of a state and (nonnegative) dense counter values. We write

$$(s, \mathbf{X}) \xrightarrow{\mathbf{m}} (s', \mathbf{X}'),$$

called a move of  $\mathcal{M}$ , if  $(s, \mathbf{m}, s') \in T$  and  $R_{\mathbf{m}}(\mathbf{X}, \mathbf{X}')$ . As usual,  $(s, \mathbf{X})$  *reaches*  $(s', \mathbf{X}')$ , written  $(s, \mathbf{X}) \rightsquigarrow (s', \mathbf{X}')$ , if there are states  $s_0 = s, s_1, \dots, s_n = s'$ , mode vectors  $\mathbf{m}_1, \dots, \mathbf{m}_n$ , counter values  $\mathbf{X}^0 = \mathbf{X}, \mathbf{X}^1, \dots, \mathbf{X}^n = \mathbf{X}'$ , for some  $n$ , such that

$$(s_0, \mathbf{X}^0) \xrightarrow{\mathbf{m}_1} (s_1, \mathbf{X}^1) \dots \xrightarrow{\mathbf{m}_n} (s_n, \mathbf{X}^n). \quad (2)$$

The set of all pairs  $(\mathbf{X}, \mathbf{X}')$  satisfying  $(s_{\text{init}}, \mathbf{X}) \rightsquigarrow (s_{\text{final}}, \mathbf{X}')$  is called the *binary reachability* of  $\mathcal{M}$ .  $\mathcal{M}$  can be thought of as a transducer, which is able to generate  $\mathbf{X}'$  from  $\mathbf{X}$  whenever  $(\mathbf{X}, \mathbf{X}')$  is in the binary reachability.

Before proceeding further, some more definitions are needed.  $\mathcal{M}$  is *monotonic* (resp. *purely dense*, *purely discrete*) if, in every transition  $(s, \mathbf{m}, s')$  in  $T$ , the mode vector  $\mathbf{m}$  does not contain any mode of unit/fractional decrement (resp. unit increment/decrement, fractional increment/decrement). Let  $r$  be a nonnegative integer. A sequence of modes is *r-reversal-bounded* if, on the sequence, the mode changes from a unit/fractional increment (followed by 0 or more stay modes) to a unit/fractional decrement and vice versa for at most  $r$  number of times. On an execution in (2) from  $s$  to  $s'$ , counter  $x_i$  is *r-reversal-bounded* if the sequence of modes for  $x_i$  is *r-reversal-bounded*. Counter  $x_i$  is *reversal-bounded* in  $\mathcal{M}$  if there is an  $r$  such that, on every execution from  $s_{\text{init}}$  to  $s_{\text{final}}$ ,  $x_i$  is *r-reversal-bounded*.  $\mathcal{M}$  is a *reversal-bounded dense multicounter machine* if every counter in  $\mathcal{M}$  is reversal-bounded.  $\mathcal{M}$  is a *reversal-bounded dense multicounter machine with a free counter* if all but one counters in  $\mathcal{M}$  are reversal-bounded. The notion of reversal-boundedness is generalized from the same notion for discrete counters [14]. Also notice that a dense counter in  $\mathcal{M}$  can be effectively restricted to be reversal-bounded, since one may use additional control states to remember each reversal and not to go over the bound. A discrete multicounter machine is a dense multicounter machine that is purely discrete and whose counters start from nonnegative integer values. Similarly, one can define a monotonic discrete multicounter machine, a reversal-bounded

discrete multicounter machine, and a reversal-bounded discrete multicounter machine with a free discrete counter (NCMF).

Let  $m$  and  $n$  be positive integers. Consider a formula

$$\sum_{1 \leq i \leq m} a_i x_i + \sum_{1 \leq j \leq n} b_j y_j \sim c,$$

where each  $x_i$  is a nonnegative real variable, each  $y_j$  is a nonnegative integer variable, each  $a_i$ , each  $b_j$  and  $c$  are integers,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$ , and  $\sim$  is  $=$ ,  $>$ , or  $\equiv_d$  for some integer  $d > 0$ . The formula is a *mixed linear constraint* if  $\sim$  is  $=$  or  $>$ . The formula is called a *dense linear constraint* if  $\sim$  is  $=$  or  $>$  and each  $b_j = 0$ ,  $1 \leq j \leq n$ . The formula is called a *discrete linear constraint* if  $\sim$  is  $>$  or  $=$ , and each  $a_i = 0$ ,  $1 \leq i \leq m$ . The formula is called a *discrete mod constraint*, if each  $a_i = 0$ ,  $1 \leq i \leq m$ , and  $\sim$  is  $\equiv_d$  for some integer  $d > 0$ .

A formula is *definable in the additive theory of reals and integers* (resp. *reals*, *integers*) if it is the result of applying quantification ( $\exists$ ) and Boolean operations ( $\neg$  and  $\wedge$ ) over mixed linear constraints (resp. dense linear constraints, discrete linear constraints); the formula is called a *mixed formula* (resp. *dense formula*, *Presburger formula*). It is decidable whether a mixed formula is satisfiable (see [21] for a quantifier elimination procedure for mixed formulas).

**Theorem 1.** *The satisfiability of mixed formulas (and hence of dense formulas and Presburger formulas) is decidable.*

A set of nonnegative real/integer tuples is definable by a mixed formula with free variables: a tuple is in the set iff the tuple satisfies the formula. Similarly, a set of nonnegative real (resp. integer) tuples is definable by a dense (resp. Presburger) formula if a tuple is in the set iff the tuple satisfies the formula. One may have already noticed that, for the purpose of this paper, our definition of mixed/Presburger/dense formulas is on nonnegative variables.

Consider a mixed formula  $R(x_1, \dots, x_n, y_1, \dots, y_m)$ . By separating each dense variable  $x_i$  into a discrete variable  $\lceil x_i \rceil$  for the integral part and a dense variable  $\lfloor x_i \rfloor$  for the fractional part,  $R(x_1, \dots, x_n, y_1, \dots, y_m)$  can always be written into the following form (after quantifier elimination), for some  $l$ :  $R_1 \vee \dots \vee R_l$ , where each  $R_i$  is in the form of

$$\exists z_1, \dots, z_n, t_1, \dots, t_n. x_1 = z_1 + t_1 \wedge \dots \wedge x_n = z_n + t_n \wedge$$

$$P(z_1, \dots, z_n, y_1, \dots, y_m) \wedge Q(t_1, \dots, t_n), \quad (3)$$

where  $z_1, \dots, z_n$  are (nonnegative) discrete variables, and  $t_1, \dots, t_n$  are dense variables (defined on the interval  $[0, 1)$ ). This representation can be easily obtained from [21]. In the representation,  $P$  is a conjunction of discrete linear constraints and discrete mod constraints and  $Q$  is a conjunction of dense linear constraints. For the given  $R$ , we define  $J(N, X_1, \dots, X_n, Y_1, \dots, Y_m)$  if there are  $x_1^i, \dots, x_n^i, y_1^i, \dots, y_m^i$ , for all  $1 \leq i \leq N$ , such that for all  $1 \leq i \leq N$ ,  $R(x_1^i, \dots, x_n^i, y_1^i, \dots, y_m^i)$  and

$$X_1 = \sum_{1 \leq i \leq N} x_1^i, \dots, X_n = \sum_{1 \leq i \leq N} x_n^i, Y_1 = \sum_{1 \leq i \leq N} y_1^i, \dots, Y_m = \sum_{1 \leq i \leq N} y_m^i.$$

Notice that all the above variables are nonnegative.



**Lemma 1.**  $J(N, X_1, \dots, X_n, Y_1, \dots, Y_m)$  is definable by a mixed formula.

Let  $R_1, \dots, R_k$  be mixed formulas over  $n$  dense variables and  $m$  discrete variables.  $\mathcal{M}$  is a monotonic multicounter machine with discrete counters  $y_1, \dots, y_m$  and dense counters  $x_1, \dots, x_n$ . All the counters start from 0.  $\mathcal{M}$  moves from its initial state and ends with its final state. On a move from one state to another,  $\mathcal{M}$  increments its counters  $(x_1, \dots, x_n, y_1, \dots, y_m)$  by any amount  $(\delta_1, \dots, \delta_n, d_1, \dots, d_m)$  satisfying  $R_i(\delta_1, \dots, \delta_n, d_1, \dots, d_m)$  – here we say  $R_i$  is picked. The choice of  $i$  is given in the description of  $\mathcal{M}$  and only depends on the source and destination state of the move. We define  $R(X_1, \dots, X_n, Y_1, \dots, Y_m)$  iff  $(X_1, \dots, X_n, Y_1, \dots, Y_m)$  are the counter values when  $\mathcal{M}$  reaches the final state.

**Theorem 2.**  $R(X_1, \dots, X_n, Y_1, \dots, Y_m)$  is definable by a mixed formula.

The following results [9] are also needed in the paper.

**Theorem 3.** *The binary reachability for a reversal-bounded discrete multicounter machine with a free counter is Presburger. Therefore, the same result also holds for a monotonic discrete multicounter machine and a reversal-bounded discrete multicounter machine.*

### 3 Decidability Results

In this section, we show a decidable characterization for various restricted versions of dense multicounter machines. The following two results can be shown using Theorem 2.

**Theorem 4.** *The binary reachability of a monotonic dense multicounter machine is definable by a mixed formula.*

**Theorem 5.** *The binary reachability of a reversal-bounded dense multicounter machine is definable by a mixed formula.*

Now, we consider a dense multicounter machine  $\mathcal{M}$  with  $k$  reversal-bounded counters  $x_1, \dots, x_k$  and a free counter  $x_0$ , in which  $s_{\text{init}}$  and  $s_{\text{final}}$  are two designated states. We further assume that  $\mathcal{M}$  satisfies the following conditions:

- (Cond1) On any execution sequence in  $\mathcal{M}$  from  $s_{\text{init}}$  to  $s_{\text{final}}$ , the reversal bounded counters are nondecreasing (i.e., never reverse),
- (Cond2) On any execution sequence in  $\mathcal{M}$  from  $s_{\text{init}}$  to  $s_{\text{final}}$ , each move will change the free counter  $x_0$  by a fractional decrement, a fractional increment, a unit decrement, or a unit increment (i.e., the free counter can not stay),
- (Cond3) On any execution sequence in  $\mathcal{M}$  from  $s_{\text{init}}$  to  $s_{\text{final}}$ , the initial and ending values of the free counter are both 0, and in between, the free counter remains positive ( $> 0$ ).

We first show a binary reachability characterization of  $\mathcal{M}$  under the three conditions. The proof of the result is rather complex. It uses a technique of shifting the free counter values so that the free counter behavior is simplified.



**Lemma 2.** *The binary reachability of reversal-bounded dense multicounter machines with a free counter, satisfying the above three conditions, is definable by a mixed formula.*

In fact, the three conditions in Lemma 2 can be completely removed, using Theorem 2 and Lemma 2.

**Theorem 6.** *The binary reachability of reversal-bounded dense multicounter machines with a free counter is definable by a mixed formula.*

Now we generalize  $\mathcal{M}$  to have *multirate* dense counters. The counters are *multirate* if a move in  $\mathcal{M}$ , in addition to the mode vector  $\mathbf{m}$ , is further associated with a  $k$ -ary positive integer vector  $\mathbf{r} = (r_1, \dots, r_k)$ . The vector  $\mathbf{r}$  is called a rate vector. The move can bring counter values from  $\mathbf{X}$  to  $\mathbf{X}'$  whenever  $R_{\mathbf{m}}(\mathbf{X}, \mathbf{X}')$ , where  $\mathbf{X}'$  is defined as follows, for each  $1 \leq i \leq k$ ,

$$\mathbf{X}''(x_i) = \frac{\mathbf{X}'(x_i) - \mathbf{X}(x_i)}{r_i}.$$

For instance, suppose  $k = 2$  and  $\mathbf{m}$  is (fractional increment, fractional decrement). Let  $\mathbf{r} = (4, 5)$ . Then the effect of the move is to increment  $x_1$  by  $4\delta$  and decrement  $x_2$  by  $5\delta$ , for some  $0 < \delta < 1$ .

One can show that Theorem 6 (and hence Theorem 4 and Theorem 5) still holds even when the dense counters are multirate, using the following ideas. Let  $\mathcal{M}$  be a reversal-bounded dense multicounter machine with a free counter. Let  $x_0, x_1, \dots, x_k$  be all the counters in  $\mathcal{M}$  ( $x_0$  is the free counter). Suppose that the reversal-bounded counters  $x_1, \dots, x_k$  are monotonic (the technique can be easily generalized). We now construct another  $\mathcal{M}'$  to simulate  $\mathcal{M}$  as follows. In  $\mathcal{M}'$ , each  $x_i$ ,  $1 \leq i \leq k$ , is replaced with many monotonic counters, namely,  $y_{\mathbf{m}}^i$ , for all mode vectors  $\mathbf{m}$ . When  $\mathcal{M}$  performs an instruction with mode vector  $\mathbf{m}$  and a rate vector  $(r_0, r_1, \dots, r_k)$ ,  $\mathcal{M}'$  performs the same instruction, repeated for  $r_0$  times, on counters  $(x_0, y_{\mathbf{m}}^1, \dots, y_{\mathbf{m}}^k)$  with mode vector  $\mathbf{m}$  but with rate  $(1, \dots, 1)$  (i.e., single rate instead of multirate). Let  $1 \leq i \leq k$ . What is the relationship between  $x_i$  in  $\mathcal{M}$  and all the  $y_{\mathbf{m}}^i$  in  $\mathcal{M}'$ ? We use  $\delta^i$  to denote the net change to counter  $x_i$  after  $\mathcal{M}$  performs the instruction. We use  $\delta_{\mathbf{m}}^i$  to denote the net change to counter  $y_{\mathbf{m}}^i$  after  $\mathcal{M}'$  performs the simulated instructions. Clearly,

$$\delta^i = \frac{r_i}{r_0} \cdot \delta_{\mathbf{m}}^i.$$

Similarly, we use  $\Delta_i$  to denote the net change to counter  $x_i$  on a path of  $\mathcal{M}$ , and use  $\Delta_{\mathbf{m}}^i$  to denote the net change to counter  $y_{\mathbf{m}}^i$  on the simulated path of  $\mathcal{M}'$ . One can show

$$\Delta_i = \frac{r_i}{r_0} \cdot \sum_{\mathbf{m}} \Delta_{\mathbf{m}}^i.$$

Since  $\mathcal{M}'$  preserves the free counter  $x_0$ , and counters in  $\mathcal{M}'$  is with single rate, from Theorem 6, we can show,

**Theorem 7.** *The binary reachability of multirate reversal-bounded dense multicounter machines (with a free counter) is definable by a mixed formula.*

So far, we have only focused on dense multicounter machines with at most one free counter. Now consider  $\mathcal{M}$  with multiple free counters. Suppose that all the counters start from 0 in  $\mathcal{M}$ . The *state reachability problem* for  $\mathcal{M}$  is: is there an execution path in  $\mathcal{M}$  from the initial state to the final state? Obviously, if  $\mathcal{M}$  contains two counters, the problem is undecidable. This is because, when the two dense counters only implement discrete increments/decrements,  $\mathcal{M}$  is a Minsky machine [17] (a two (discrete) counter machine). A more interesting case is when  $\mathcal{M}$  contains *purely dense* free counters only; i.e., the counters do not perform any discrete increments/decrements. The following theorem states that the state reachability problem is undecidable even when  $\mathcal{M}$  has only four purely dense free counters.

**Theorem 8.** *The state reachability problem for purely dense multicounter machines is undecidable. The undecidability remains even when there are only four purely dense free counters.*

Turning now to the case when  $\mathcal{M}$  has only two purely dense free counters, we have the following decidable result. The case when there are three counters is open.

**Theorem 9.** *The state reachability problem for machines with only two purely dense free counters is decidable.*

## 4 Safety/Liveness Verification

From the binary reachability characterizations given in the previous section, one can establish decidable results on various verification problems for a reversal-bounded dense multicounter machine  $\mathcal{M}$  with a free counter, even when counters in  $\mathcal{M}$  are of multirate.

The *binary reachability problem* for  $\mathcal{M}$  is defined as follows: Given a mixed formula  $I(\mathbf{X}, \mathbf{X}')$ , are there  $\mathbf{X}$  and  $\mathbf{X}'$  such that  $I(\mathbf{X}, \mathbf{X}')$  holds and  $(s_{\text{init}}, \mathbf{X})$  reaches  $(s_{\text{final}}, \mathbf{X}')$  in  $\mathcal{M}$ ? An example of the problem is as follows:

Is there  $(x_1, x_2, x_3, x'_1, x'_2, x'_3)$  satisfying “ $x'_1 - 2x_1 + x_2$  is an integer  $\wedge x'_3 + x'_2 > x_3 + x_2 + x_1 + 1$ ” and  $(s_{\text{init}}, x_1, x_2, x_3)$  reaches  $(s_{\text{final}}, x'_1, x'_2, x'_3)$  in  $\mathcal{M}$ ?

Using Theorem 1 and Theorem 6, one can show that the binary reachability problem is decidable.

Additionally, one can also consider the *mixed i.o. (infinitely often) problem* for  $\mathcal{M}$  as follows. Given two mixed formulas  $I$  and  $P$ , the problem is to decide whether there is an infinite execution path of  $\mathcal{M}$

$$(s^0, \mathbf{X}^0) \rightarrow (s^1, \mathbf{X}^1) \rightarrow \dots \rightarrow (s^n, \mathbf{X}^n) \rightarrow \dots$$

such that  $s^0 = s_{\text{init}}$ ,  $I(\mathbf{X}_0)$  holds, and there are infinitely  $n_1 < n_2 < \dots$  such that  $P(\mathbf{X}^{n_i})$  holds for all  $i$ . An example of the problem is as follows: Is there an infinite run of  $\mathcal{M}$  such that

$$x_3 > x_2 + x_1 - 1 \wedge 2x_2 - x_1 \text{ is an integer}$$

is satisfied for infinitely many times? The problem can be used to formalize some fairness properties along an  $\omega$  execution path of a transition system (see also [11]). Because of

Theorem 6,  $\mathcal{M}$  is a mixed linear system in the sense of [10]. From the main theorem in [10], one can conclude that the mixed i.o. problem is decidable.

A dense monotonic counter machine with a free counter (with multirate counters) can be used to model a dense counter system with counter instructions in the following form

$$\exists 0 < \delta < 1. x'_0 = x_0 + r_0 \cdot t_0 \wedge x'_1 = x_1 + r_1 \cdot t_1 \wedge \cdots \wedge x'_k = x_k + r_k \cdot t_k,$$

where,  $r_0, \dots, r_k$  are positive integers (rates),  $x_0$  is the free counter, and  $x_1, \dots, x_k$  are the monotonic counters. We use the primed variables to indicate the new values. The term  $t_0$  for the free counter  $x_0$  is one of  $0, \delta, -\delta, 1, -1$ . The term  $t_i, 1 \leq i \leq k$ , for monotonic counter  $x_i$  is one of  $0, \delta, 1$ . In addition, the counters can be tested against 0. Though the machine models are intended to be a fundamental tool to reason about a broader class of dense counter systems, using Theorem 6, one can also use the model to specify some interesting systems with multiple dense/discrete counters (e.g., the consumer/producer system in Section 1).

## 5 Discussions

It is a natural idea to transform a dense counter machine  $\mathcal{M}$  into a dense counter automaton (i.e., acceptor)  $\mathcal{A}$  in which a one-way input tape is provided. In contrast to a traditional word automaton, a dense word is provided on the input tape. We elaborate on this as follows. A *block*  $B$  is a pair  $(c, \delta)$ , where  $c \in C$  is the color of the block and  $0 \leq \delta \leq 1$  is called the block's *length*. There are only finitely many possible colors in  $C$ . Let *Blocks* be the set of all blocks on colors in  $C$  and *Blocks*<sup>\*</sup> be the free monoid generated by the infinite set *Blocks*. Hence, there exists an associative operation (called concatenation) of blocks with an identity  $\epsilon$ . The Kleene closures <sup>\*</sup> and <sup>+</sup> are defined as usual using concatenation. A *dense word* is an element of *Blocks*<sup>\*</sup> and a *dense language* is a subset of *Blocks*<sup>\*</sup>. Given a dense word  $w$  as input,  $\mathcal{A}$  reads the blocks in  $w$  one by one. On reading a block,  $\mathcal{A}$  updates its control state and the counters in the same way as  $\mathcal{M}$  does, except that  $\mathcal{A}$  knows the color of the block, can distinguish whether the length of the block is 0, 1, or strictly greater than 0 and strictly less than 1.  $\mathcal{A}$  can increment/decrement some of the counters by 0, 1, or the length of the block. In this way, one can similarly define dense monotonic counter automata, dense reversal-bounded counter automata, and dense reversal-bounded counter automata with a free counter.

For a dense word  $w$ , we use  $\#_c(w)$  (resp.  $l_c(w)$ ) to denote the total number (resp. length) of blocks with color  $c$  in  $w$ . We use  $Parikh(w)$  to denote the tuple of counts  $\#_c(w)$  and  $l_c(w)$  for each  $c \in C$ ; i.e.,

$$Parikh(w) = (\#_{c_1}(w), \dots, \#_{c_n}(w), l_{c_1}(w), \dots, l_{c_n}(w)).$$

For a dense language  $L$ , we use  $Parikh(L)$ , the Parikh map of  $L$ , to denote the set of all  $Parikh(w)$  for  $w \in L$ . A dense language  $L$  is a *mixed semilinear* dense language if  $Parikh(L)$  is definable by a mixed formula  $F$  over  $n$  integer variables and  $n$  dense variables: For all  $x_1, \dots, x_n \in \mathbb{N}$  and for all  $y_1, \dots, y_n \in \mathbb{R}$ ,

$$(x_1, \dots, x_n, y_1, \dots, y_n) \in Parikh(L)$$

iff  $F(x_1, \dots, x_n, y_1, \dots, y_n)$  holds. The above definition of a Parikh map for a dense language can be seen as a natural extension of the concept of Parikh maps for discrete languages. Indeed, one can treat a discrete language  $\mathcal{L}$  as a special dense language  $L_{\mathcal{L}}$ , where a symbol in  $C$  is understood as a unit block. One can easily verify that  $\mathcal{L}$  is a semilinear discrete language iff  $L_{\mathcal{L}}$  is a mixed semilinear dense language. So, over discrete languages, our definition of “mixed semilinearity” coincides with the traditional “semilinearity” definition.

It is straightforward to verify the following theorem, using the results in Section 3.

**Theorem 10.** *Dense languages accepted by dense reversal-bounded counter automata with a free counter are mixed semilinear languages. Hence, the emptiness and infiniteness problems for these automata are decidable.*

One can define a deterministic dense reversal-bounded counter automaton in the usual way, with the additional requirement that the transition the automaton makes on an input block  $(c, \delta)$  with  $0 < \delta < 1$  should be the same if the block is replaced by any  $(c, \delta')$  with  $0 < \delta' < 1$ .

The following corollaries are easily verified:

**Corollary 1.** *Let  $\mathcal{A}_1$  and  $\mathcal{A}_2$  be dense reversal-bounded counter automata.*

1. *We can effectively construct dense reversal-bounded counter automata  $\mathcal{A}$  and  $\mathcal{A}'$  accepting  $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$  and  $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ , respectively.*
2. *If  $\mathcal{A}_1$  is deterministic, we can effectively construct a deterministic dense reversal-bounded counter automaton accepting the complement of  $L(\mathcal{A}_1)$ .*

*If one of  $\mathcal{A}_1$  or  $\mathcal{A}_2$  (but not both) has a free counter, then  $\mathcal{A}$  and  $\mathcal{A}'$  will also have a free counter. It follows that the class of languages accepted by deterministic dense reversal-bounded counter automata is closed under the boolean operations.*

**Corollary 2.** *The emptiness, infiniteness, and disjointness problems for dense reversal-bounded counter automata are decidable. The containment and equivalence problems for deterministic dense reversal-bounded counter automata are decidable.*

Notice that the universe problem for dense reversal-bounded counter automata is undecidable, since the undecidability holds even for discrete counters [14]. Obviously, from Theorem 10, when the automata (even with a free counter) are deterministic, the universe problem is decidable.

## References

1. R. Alur, C. Courcoubetis, and T. A. Henzinger. Computing accumulated delays in real-time systems. *Formal Methods in System Design: An International Journal*, 11(2):137–155, August 1997.
2. R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

3. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
4. B. Boigelot, S. Rassart, and P. Wolper. On the expressiveness of real and integer arithmetic automata. In *ICALP'98*, volume 1443 of *LNCS*, pages 152–163. Springer, 1998.
5. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV'94*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.
6. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *CAV'98*, volume 1427 of *LNCS*, pages 268–279. Springer, 1998.
7. H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *CONCUR'99*, volume 1664 of *LNCS*, pages 242–257. Springer, 1999.
8. Z. Dang, O. H. Ibarra, and Z. Sun. On the emptiness problems for two-way nondeterministic finite automata with one reversal-bounded counter. In *ISAAC'02*, volume 2518 of *LNCS*, pages 103–114. Springer, 2002.
9. Zhe Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. In *CAV'00*, volume 1855 of *LNCS*, pages 69–84. Springer, 2000.
10. Zhe Dang and Oscar H. Ibarra. On the existence of  $\omega$ -chains for transitive mixed linear relations and its applications. *International Journal of Foundations of Computer Science*, 13(6):911–936, 2002.
11. Zhe Dang, P. San Pietro, and R. A. Kemmerer. On Presburger Liveness of Discrete Timed Automata. In *STACS'01*, volume 2010 of *LNCS*, pages 132–143. Springer, 2001.
12. L. Fribourg and H. Olsen. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4):305–335, 1997.
13. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In *CAV'97*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.
14. O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, January 1978.
15. O. H. Ibarra, T. Jiang, N. Tran, and H. Wang. New decidability results concerning two-way counter machines. *SIAM J. Comput.*, 24:123–137, 1995.
16. K. Larsen, G. Behrmann, Ed Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *CAV'01*, volume 2102 of *LNCS*, pages 493–505. Springer, 2001.
17. M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74:437–455, 1961.
18. X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178, 1992.
19. A. Puri, P. Kopke, T. Henzinger and P. Varaiya. What's decidable about hybrid automata? *27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 372–382, 1995.
20. P. Z. Revesz. A closed form for datalog queries with integer order. *Proc. Intl. Conf. on Database Theory (ICDT'90)*, volume 470 of *LNCS*, pages 187–201. Springer, 1990.
21. V. Weispfenning. Mixed real-integer linear quantifier elimination. *Proc. Intl. Symp. on Symbolic and Algebraic Computation*, pages 129–136, Vancouver, B.C., Canada, July 29–31, 1999.

# TRIM: A Tool for Triggered Message Sequence Charts<sup>\*</sup>

Bikram Sengupta and Rance Cleaveland

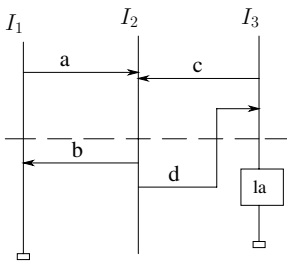
Department of Computer Science, SUNY at Stony Brook  
Stony Brook, NY 11794-4400, USA  
{sbikram,rance}@cs.sunysb.edu

**Abstract.** TRIM is a tool for analyzing system requirements expressed using *Triggered Message Sequence Charts* (TMSCs). TMSCs enhance MSCs with capabilities for expressing conditional and partial behavior and with a refinement ordering. This paper shows how the Concurrency Workbench of the New Century may be adapted to check refinements between TMSC specifications.

## 1 Introduction

Triggered Message Sequence Charts (TMSCs) [14] are a scenario-based visual formalism for capturing requirements of distributed systems. TMSCs enhance traditional MSCs [3] with capabilities for expressing *conditional* and *partial* behavior and with a mathematically precise notion of *refinement*, which may be used to check whether one set of requirements correctly elaborates on another. This paper presents TRIM, a tool for checking refinement between TMSCs. The main features of TRIM are: (i) a textual language for TMSCs that includes the algebraic combinators of [14]; (ii) a routine for checking refinements among TMSC specifications; and (iii) a capability for generating diagnostic information in the form of *tests* when one system fails to refine another.

**TMSCs.** Graphically, TMSCs, as exemplified in Fig. 1, extend MSCs in two respects.



**Fig. 1.** An Example TMSC

The first is the dashed horizontal line cutting across the *instances* (vertical axes) and partitioning the sequences of events for each instance into a *trigger* — the subsequence above the line — and an *action* — the subsequence below. A TMSC requires that if an instance performs its trigger, then it must execute its action; otherwise, it is unconstrained. The second new feature in TMSCs is the possibility of a hollow bar at the foot of each instance, as in instances  $I_1$  and  $I_3$  in Fig. 1, and whose presence signals *termination*: no further behavior is allowed. A bar's absence (cf. instance  $I_2$ ) means that there are no constraints on subsequent behavior, which may be extended in the future. TMSCs

have an abstract textual syntax that is described formally in [14]; [15] compares and contrasts TMSCs with other MSC-based formalisms.

<sup>\*</sup> Research supported by NSF grants CCR-9988489 and CCR-0098037 and Army Research Office grants DAAD190110003 and DAAD190110019.

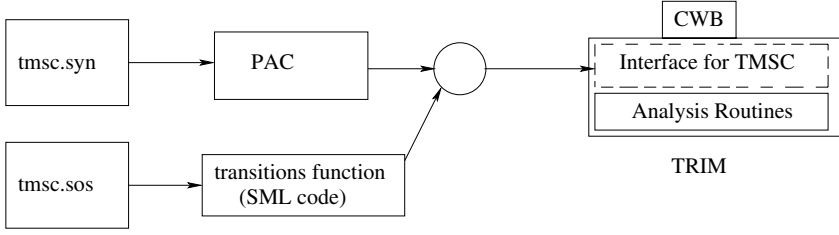


Fig. 2. Implementing TRIM

**TMSC Expressions.** Single TMSCs capture individual system requirements. The work in [14] also presents a set of operators for constructing structured collections of TMSCs. The resulting *TMSC expressions* have the following syntax:

$S ::= M$	(single TMSC)	$X$	(variable)
$  S; S$	(sequential composition)	$S \parallel S$	(parallel composition)
$  S \nabla S$	(delayed choice)	$recX.S$	(recursive operator)
$  S \oplus S$	(internal choice)	$S \wedge S$	(logical and)

The language includes programming-like constructs such as sequential and parallel composition, delayed choice, and recursion (to express iterative behavior), as well as more declarative operators such as internal choice and conjunction. The latter enable TMSCs to capture logical as well as operational requirements. The semantics of TMSC expressions is based on *acceptance tress* and the *must preorder* [12,14].

## 2 TRIM

TRIM supports the textual notation for TMSCs given in [14] and provides: **a simulator** for executing TMSC expressions; **a compilation tool** for converting TMSC expressions into manually inspectable acceptance trees for manual inspection (impractical for large examples but often effective for smaller, early-stage artifacts); **routines for checking the refinement ordering** between TMSC expressions, and for returning diagnostic information when refinement fails to hold. The tool also includes routines for checking temporal properties of, and minimizing, finite-state TMSC expressions. The remainder of this section briefly describes how TRIM is implemented and used.

**Implementing TRIM.** TRIM is implemented on top of the Concurrency Workbench of the New Century (CWB-NC) [8,9], an easy-to-retarget verification tool for finite-state systems. Instances of the CWB-NC consist of a front end that handles syntax and semantic issues of design notations, and a back-end that implements the analysis routines, including a simulator, a model checker, and several refinement-checking procedures. As the CWB-NC computes the must preorder, a natural approach for TRIM is to develop a TMSC front end for the CWB-NC. This is what we did; Fig. 2 gives an overview. The remainder of this section describes some of the issues involved in this program.

CWB-NC front ends must contain: a parser, an unparser, and a routine for computing the single-step transitions of system descriptions. The Process Algebra Compiler (PAC) [7] is designed to simplify the task of implementing CWB-NC front ends. The PAC generates front ends from design-language specifications describing the syntax of the language in the form a YACC-like grammar and the semantics of the language given as sets of Plotkin-style SOS rules [10]. The two specifications for a given language  $L$  are stored in two different files:  $L.syn$  for the syntax and  $L.sos$  for the semantics. Our initial goal was to use the PAC to generate the TMSC front end, and to this end we needed to devise two files:  $tmsc.syn$  and  $tmsc.sos$ .

Implementing  $tmsc.syn$  was straightforward. Devising SOS rules proved trickier. The semantics in [14] is essentially denotational; it defines the meaning of TMSC expression operators as constructions mapping acceptance trees to acceptance trees. It also allows for an arbitrary number of messages to be “pending”, i.e. sent but not yet received. Both of these features pose problems for the CWB-NC, which requires an operational rather than a denotational semantics and also needs semantic entities to be *finite-state*. To cope with the former problem we gave SOS rules that are provably equivalent to the original declarative semantics; the main subtlety involved handling the interplay between nondeterminism and the conjunction operator. To help with the latter we equipped the operational semantics with a parameter that can be used to bound the number of messages in transit (i.e. the buffer size). This semantics thus approximates the “true” semantics, although they coincide for TMSC expressions whose number of pending messages never exceeds the parameter in the operational semantics.

We could not use the PAC to generate the semantic functions directly from our SOS rules, owing to PAC restrictions. For example, PAC does not allow the definition of mutually recursive auxiliary semantic relations; and yet our treatment of conjunction required this. Hence, we wrote the *transitions* function by hand from the  $tmsc.sos$  file, producing around 2,500 lines of SML code, and integrated it with the PAC-generated parser to build the TRIM interface for the TMSC language; see Fig. 2.

**Using TRIM.** TRIM is a research prototype; we did not pay close attention to performance issues in implementing the front end. Nevertheless, we have used TRIM to process several different TMSC-based specifications, including a simple protocol for atomic reading and writing [14]; the specification of an automated infusion pump used in treating trauma patients [2,13]; the well-known steam-boiler example in [4,13]; and a component of an air-traffic control system [1,15]. The resulting transition systems ranged in size from 1,744 to 26,183 states, with the corresponding acceptance trees containing from 196 to 2,495 nodes. In all cases, refinements were proposed in terms of more deterministic TMSC expressions, and were verified using TRIM. The counter-example generation feature of CWB was a major advantage, especially because TRIM currently supports only a text-based interface and complex TMSC expressions typed in by a user are subject to typographical errors such as unintentional mistakes in message names. The analysis times ranged from several minutes to several hours, although performance improvements of two orders of magnitude are achievable, in our view.

**Related Work.** Several tools have been developed to support the use of scenarios in design requirements. MESA [6] allows certain properties, such as process divergence to



be efficiently checked on MSCs. UBET ([5]) detects potential race conditions and timing violations in an MSC, and also provides automatic test case generation over HMSCs. The *play-in/play-out* approach of [11] is based on LSCs and has been implemented via a tool called the *play engine*. The tool LTSA-MSC [16] supports the synthesis of behavior models from MSC-based specifications and implied-scenario detection.

### 3 Conclusions and Future Work

We have presented TRIM, a tool that provides automated support for analyzing system requirements given in the TMSC notation [14]. The tool provides a number of useful routines, including a simulator and a refinement checker, that are inherited from the CWB-NC verification tool on which it is based. Retargeting the CWB-NC to TMSCs required us to adapt the semantic account of the notation given in [14] to (i) make it operational and (ii) to bound pending messages. For future work, we plan to improve the performance of the TRIM front end and to develop a graphical user interface.

### References

1. Center-TRACON automation system (CTAS). URL:<http://ctas.arc.nasa.gov/>.
2. Integrated Medical Systems Inc. URL:<http://www.lstat.com/lstat.html>.
3. Message sequence charts (MSC). *ITU-TS Recommendation Z.120*, 1996.
4. J. R. Abrial, E. Börger, and H. Langmaack. Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. *LNCS volume 1165*, 1996.
5. R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.
6. H. Ben-Abdallah and S. Leue. MESA: Support for scenario-based design of concurrent systems. *Proc. TACAS'98*, LNCS volume 1384:118–135.
7. R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. *Proc. TACAS'95*, LNCS volume 1019:153–173.
8. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1):36–72, 1993.
9. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42(1):39–47, January 2002.
10. G. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.
11. D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. *Software and System Modeling (SoSym)*, 2003.
12. M. Hennessy. Algebraic theory of processes. *The MIT Press*, 1988.
13. B. Sengupta and R. Cleaveland. Refinement-based requirements elicitation using triggered message sequence charts. To appear in 2003 Intl. Requirements Engineering Conf.
14. B. Sengupta and R. Cleaveland. Triggered message sequence charts. *Proceedings of ACM SIGSOFT 2002, FSE-10*.
15. B. Sengupta and R. Cleaveland. Towards formal but flexible scenarios. *2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, at ICSE 2003.
16. J. Kramer, S. Uchitel, and J. Magee. LTSA-MSC: Tool support for behaviour model elaboration using implied scenarios. *Proc. TACAS'03*.

# Model Checking Multi-Agent Programs with CASP

Rafael H. Bordini<sup>1</sup>, Michael Fisher<sup>1</sup>, Carmen Pardavila<sup>1</sup>,  
Willem Visser<sup>2</sup>, and Michael Wooldridge<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, U.K.  
{R.Bordini, M.Fisher, C.Pardavila, M.J.Wooldridge}@csc.liv.ac.uk

<sup>2</sup> RIACS/NASA Ames Research Center, Moffett Field, CA 94035, U.S.A.  
wvisser@email.arc.nasa.gov

## 1 Introduction

In order to provide generic development tools for rational agents, a number of agent programming languages are now being developed, often by extending conventional programming languages with capabilities from the BDI (Belief-Desire-Intention) theory of rational agency [7,9]. Such languages provide high-level abstractions that aid the construction of dynamic, autonomous components, together with the deliberation that goes on within them. One particularly influential example of such a language is AgentSpeak(L) [6], a logic programming language with abstractions provided for key aspects of rational agency, such as beliefs, goals and plans.

Model checking techniques have only recently begun to find a significant audience in the multi-agent systems community. In particular, our approach is the first whereby model checking can be applied to a logic programming language aimed at reactive planning systems following the BDI architecture.

Our aim in this paper is to describe a toolkit we have developed to support the use of model checking techniques for AgentSpeak(L). The toolkit, called CASP (Checking AgentSpeak Programs), automatically translates AgentSpeak(L) code into the input language of existing model checkers. In [1], we showed how to translate from AgentSpeak(L) to PROMELA, the model specification language for the SPIN LTL model checker [5]. More recently [2], we developed an alternative approach, based on the translation of AgentSpeak(L) agents into Java and verification via JPF2, a general purpose Java model checker [8].

## 2 AgentSpeak(L)

The AgentSpeak(L) programming language was introduced in [6]. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to implementing “intelligent” or “rational” agents [9]. An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A *belief atom* is simply a first-order predicate in

the usual notation, and belief atoms or their negations are termed *belief literals*. An *initial set of beliefs* is just a collection of ground belief atoms.

AgentSpeak(L) distinguishes two types of goals: *achievement goals* and *test goals*. Achievement and test goals are predicates (as for beliefs) prefixed with operators ‘!’ and ‘?’ respectively. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of *subplans*.) A *test goal* returns a unification for the associated predicate with one of the agent’s beliefs; they fail otherwise. A *triggering event* defines which events may initiate the execution of a plan. An *event* can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. There are two types of triggering events: those related to the *addition* (‘+’) and *deletion* (‘-’) of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols used to distinguish them. If  $e$  is a triggering event,  $b_1, \dots, b_m$  are belief literals, and  $h_1, \dots, h_n$  are goals or actions, then “ $e : b_1 \ \& \ \dots \ \& \ b_m \leftarrow h_1 ; \dots ; h_n$ .” is a *plan*. A plan is formed by a *triggering event* (denoting the purpose for that plan), followed by a conjunction of belief literals representing a *context* (they are separated by ‘:’). The context must be a logical consequence of that agent’s current beliefs for the plan to be *applicable*. The remainder of the plan (after ‘ $\leftarrow$ ’) is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

Figure 1 shows some example AgentSpeak(L) plans. They tell us that, when a concert is announced for artist  $A$  at venue  $V$  (so that, from perception of the environment, a belief  $\text{concert}(A, V)$  is *added*), then if this agent in fact likes artist  $A$ , then it will have the new goal of booking tickets for that concert. The second plan tells us that whenever this agent adopts the goal of booking tickets for  $A$ ’s performance at  $V$ , if it is the case that the telephone is not busy, then it can execute a plan consisting of performing the

```
+concert(A,V) : likes(A)
  <- !book_tickets(A,V).

+!book_tickets(A,V)
  : not(busy(phone))
  <- call(V); ...;
    !choose_seats(A,V).
```

**Fig. 1.** Examples of Plans

basic action  $\text{call}(V)$  (assuming that making a phone call is an atomic action that the agent can perform) followed by a certain protocol for booking tickets (indicated by ‘...’), which in this case ends with the execution of a plan for choosing the seats for such performance at that particular venue.

### 3 Property Specification Language

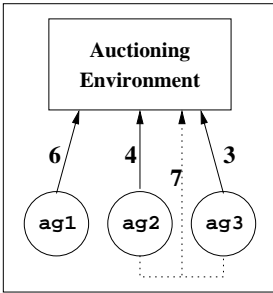
In the context of verifying multi-agent systems implemented in AgentSpeak(L), the most appropriate way of specifying the properties that the system satisfies (or does not satisfy) is expressing those properties within BDI logics [7,9]. In our framework, we can express simple BDI logical properties that can be subse-

quently translated into Linear Temporal Logic (LTL) formulæ (as used by SPIN and JPF2) with associated predicates over AgentSpeak(L) data structures.

Our property specification language includes the standard BDI modal operators *Bel* (Belief), *Des* (Desire), and *Int* (Intention); however, these can only be applied to AgentSpeak(L) atomic formulæ. The language also includes a modality used to refer to an agent performing an action in the environment (called *Does*). For example, for an agent *i* as in Figure 1, one can express properties such as  $\Box((\text{Des } i \text{ book\_tickets}(a1, v1)) \Rightarrow (\text{Bel } i \text{ likes}(a1)))$  and  $\Box((\text{Int } i \text{ book\_tickets}(a1, v1)) \Rightarrow \Diamond(\text{Does } i \text{ call}(v1)))$ . Note that an intention requires an applicable plan; in BDI theory, intentions are desired states of affair which an agents has committed itself to achieve (in practice, through the execution of a plan). Further details of the language can be found in [1].

## 4 Practical Model Checking

In [1], we defined a finite-state version of AgentSpeak(L), called AgentSpeak(F), and we showed how to convert a set of AgentSpeak(F) programs into PROMELA, as well as how to convert BDI properties into LTL formulæ (following the translation approach mentioned above). We can then use the SPIN model checker to verify multi-agent systems written in AgentSpeak(F). Recently, we introduced an alternative approach where AgentSpeak(F) programs are translated to Java code, thus allowing the use of JPF2 for model checking [2]. Note that before model checking can start, one also needs to encode the environment where the agents are to be situated in the input notation of the model checker being used.



**Fig. 2.** Auction System

One of the case studies we carried out to assess our approach was the analysis of a simplified auction scenario, illustrated in Figure 2. A simple environment announces 10 auctions and states which agent is the winner in each one (the one with the highest bid). There are three agents, written in AgentSpeak(F), participating in these auctions. Agent *ag1* is a very simple agent which bids 6 whenever the environment announces a new auction. Agent *ag2* bids 4, unless it has agreed on an alliance with *ag3*, in which case it bids 0. Agent *ag3* tries to win the first *T* auctions, where *T* is a threshold stored in its belief base. If it does not win any auctions up to that point, it will try to achieve an alliance with *ag2* (by sending the appropriate message to it). When *ag2* confirms that it agrees to form an alliance, then *ag3* starts bidding, on behalf of them both, the sum of their usual bids (i.e., 7).

Initial results have indicated that, while Java provides a much more appropriate target language than PROMELA, JPF2 does not scale as well as SPIN. Java is the language of choice in most practical implementations of multi-agent systems, and the Java model is much more clear and easily extensible; JPF2 also handles unbounded data structures, so we do not have to limit them during translation time. We have used both model checkers for verifying that the system described above satisfies the following specifications (among others):

- (i)  $\Box( \neg(\text{Bel ag3 winner(ag3)}) \wedge (\text{Des ag3 alliance(ag3, ag2)}) \Rightarrow \Diamond(\text{Int ag3 alliance(ag3, ag2)}) );$
- (ii)  $\Diamond( (\text{Bel ag2 alliance(ag3, ag2)}) \wedge (\text{Bel ag3 alliance(ag3, ag2)}) );$  and
- (iii)  $\Box( (\text{Bel ag2 alliance(ag3, ag2)}) \wedge (\text{Bel ag3 alliance(ag3, ag2)}) \Rightarrow \Diamond\Box\text{winner(ag3)} ).$

## 5 Ongoing and Future Work

We are currently attempting to improve the efficiency of the AgentSpeak(F) models by optimisations on the PROMELA or Java code that is automatically generated. We are also working on a deeper analysis of the advantages and disadvantages of those model checkers in the verification of AgentSpeak(F) systems.

As future work, we intend to examine symbolic model checking for AgentSpeak(F), possibly by using NuSMV2 [3]. We also plan to combine our present approach with deductive verification so that we can handle larger applications. Further, it would be interesting to add extra features to our approach to agent verification (e.g., handling plan failure, allowing first order terms, allowing variables in the specifications). Finally, we also plan as future work to verify more ambitious applications, such as autonomous spacecraft control (along the lines of [4]).

**Acknowledgements:** This work has been partially supported by a Marie Curie fellowship of the EC, contract HPMF-CT-2001-00065 (“Model Checking for Mobility”).

## References

1. R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. *2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, 2003.
2. R. H. Bordini, W. Visser, M. Fisher, and M. Wooldridge. Model checking a reactive planning language for multi-agent systems. Submitted, 2003.
3. A. Cimatti *et al.*. NuSMV2: an opensource tool for symbolic model checking. *14th Int. Conf. on Computer Aided Verification*, LNCS 2404, Springer-Verlag, 2002.
4. M. Fisher and W. Visser. Verification of autonomous spacecraft control — a logical vision of the future. *Workshop on AI Planning and Scheduling For Autonomy in Space Applications, co-located with TIME*, 2002.
5. G. J. Holzmann. The Spin model checker. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997.
6. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. *7th MAAMAW Workshop*, LNAI 1038, London, 1996. Springer-Verlag.
7. A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.
8. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *15th Int. Conf. on Automated Software Engineering*. IEEE Computer Society, 2000.
9. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.

# Monitoring Temporal Rules Combined with Time Series

Doron Drusinsky<sup>1,2</sup>

<sup>1</sup> Naval Postgraduate School, Monterey, CA, USA  
ddrusins@nps.navy.mil

<sup>2</sup> Time-Rover, Inc., 11425 Charsan Ln., Cupertino, CA 95014, USA  
doron@time-rover.com, www.time-rover.com

**Abstract.** Run-time monitoring of temporal properties and assertions is used for testing and as a component of execution-based model checking techniques. Traditional run-time monitoring however, is limited to observing sequences of pure Boolean propositions. This paper describes tools, which observe temporal properties over time series, namely, sequences of propositions with constraints on data value changes over time. Using such temporal logic with time series (LTL<sub>D</sub>) it is possible to monitor important properties such as stability, monotonicity, temporal average and sum values, and temporal min/max values. The paper describes the Temporal Rover and the DBRover, which are in-process and remote run-time monitoring tools, respectively, that support linear time temporal logic (LTL) with real-time (MTL) and time series (LTL<sub>D</sub>) constraints.

## 1. Temporal Logic and Run-Time Monitoring Overview

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. In [6], Pnueli suggested using Linear-Time Propositional Temporal Logic (LTL) for reasoning about concurrent programs. Since then, several researchers have used LTL to state and prove correctness of concurrent programs, protocols, and hardware.

Linear-Time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the propositional logic operators there are four future-time operators and four dual past time operators: always in the future (always in the past), eventually, or sometime in the future (sometime in the past), until (Since), and next cycle (previous cycle). Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [1]. MTL extends LTL by supporting the specification of relative time and real time constraints. All four LTL future time operators can be constrained by relative time and real time constraints specifying the duration of the temporal operator. This paper described additional extension to LTL and MTL suitable for the specification of time-series constraints.

Run time Execution Monitoring (REM) is a class of methods of tracking temporal requirements for an underlying application. First applications of REM were verification oriented where REM methods were used to track whether an executing system conforms to formal specification requirements. Recent adaptations of REM methods enable run time monitoring for non-verification purposes such as temporal business rule checking and temporal security rule checking [5]. Unlike previously published

methods [7], the newer methods are *on-line*, namely, temporal rules are evaluated without storing an ever growing and potentially unbounded history trace. The TemporalRover and DBRover tools described in this paper perform on-line REM using executable alternating finite automata. The technique enables on-line monitoring complex Kansas State Specification Pattern assertions at a rate of 6000 to 60,000 cycles per second on a 1GHz CPU [4], and is capable of monitoring past-time and future-time temporal logic augmented with real-time constraints, time-series constraints, and special counting operators described in [2]. High-speed on-line REM enables demanding applications such as formal specification based exception handling [3].

## 2. Run Time Monitoring Tools: The Temporal Rover and DBRover

The Temporal Rover [2] is a code generator whose input is a Java, C, C++, or HDL source code program, where LTL/MTL assertions are embedded as source code comments. The Temporal Rover parser converts this program file into a new file, which is identical to the original file except for the assertions that are now implemented in source code. The following example contains an embedded MTL assertion for a Traffic Light Controller (TLC) written using the Temporal Rover syntax asserting that *for 100 milliseconds, whenever light is red, camera s.b. on*:

```
void tlc(int Color_Main, boolean CameraOn) {
... /* Traffic Light Controller functionality */
/* TRBegin
   TRClock{C1=getTimeInMillis()} // get time from OS
   TRAssert{ Always({Color_Main == RED} Implies
                    Eventually_C1<1000_{CameraOn == 1})
              } =>
           //      Customizable      user      actions
   {printf("SUCCESS");printf("FAIL");printf("DONE!");}
   TREnd */
} /* end of tlc */
```

The TemporalRover generates code that replaces the embedded LTL/MTL assertion with real C, C++, Java, or HDL code, which executes in-process, i.e., as part of the underlying application. The DBRover is a remote monitor version of the TemporalRover whereby assertions are monitored on a remote machine, using HTTP, sockets, or serial communication with the underlying target application.

## 3. LTL and MTL with Time Series Constraints (LTLD)

While LTL and MTL assert about sequences of pure Boolean propositions, it is often required to assert about sequences of propositions over time series, i.e., series of data values with constraints on the change of those values over time. For example, consider a requirement  $R$ , stating that *for one minute as of eventA, the value of variable  $x$  should be 10% stable*. Such a requirement combines MTL with propositions

based on temporal instances of a variable  $x$ . The need for such time series assertions typically involves the validation of statistical and algebraic artifacts such as stability, monotonicity, averaging and expectancy, sum and product values, time-series properties, min/max values, etc. Specific examples of such time series assertions are listed in the sequel.

Like LTL, LTL augmented with time series (LTLD) assertions are non-deterministic and might have multiple overlapping instances active simultaneously. For example, in requirement  $R$  above, the values of a same variable name  $x$  are referred to and compared with one another in multiple points in time, for a plurality of *eventA*'s, i.e., for a plurality of initial  $x$  values. One of many possible scenarios is where *eventA* occurs first when  $x=100$ , and then occurs again 30 seconds later when  $x=110$ ; hence, in the overlapping 30 second time-segment,  $x$  values must range between 99 and 110. Clearly, the number and timing of *eventA* occurrences is unknown in advance, and the simple 1-minute end condition could, in general, be non-deterministic, rendering the task of monitoring all possible scenarios non-trivial.

LTLD enables the specification of requirements in which propositions include temporal instances of variables. Consider the following *automotive cruise control* code with an embedded stability assertion requiring speed to be 5% stable while cruise is set and not changed (uses TemporalRover syntax):

```
Void cruise(boolean cruiseSet, boolean cruiseChange,
boolean cruiseOff, boolean cruiseIncr, int speed){
    ... /* Cruise Controller functionality */
    /* TRBegin
        TRAssert{Always ({cruiseSet}Implies
            {speed*0.95<speed' && speed'<speed*1.05}
            Until $speed$
            {cruiseChange || cruiseOff}
            ) }=> {...} // user actions

    TREnd */
```

In this example speed is a temporal data variable, which is associated with the Until temporal operator. This association implies that every time the Until operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in speed variable of this instance of the *Until* operator; this value is referred to as the pivot value for this *Until* operator instance. Future speed values used by this particular evaluation of the *Until* statement are referred to using the prime notation, i.e., as *speed'*, and are referred to as primed values. Hence, if speed is 100Kmh when *cruiseSet* is true, then the pivot value for speed is 100, while every subsequent speed value is referred to as *speed'* and must be within 5% of the (pivot) speed.

Note how *speed* is declared using the *\$speed\$* notation to be a temporal data variable associated with the *Until* operator. This declaration indicates to the Temporal Rover that it should be sampling a pivot value from the environment in the first cycle of the *Until* operators lifecycle, and to refer to all subsequent samples of speed as *speed'*.

Similarly, the following example consists of a *monotonicity* requirement for the cruise control system, where speed is monotonically increasing while Cruise Increase (*cruiseIncr*) command is active:



```

TRAssert {Always ({cruiseIncr}Implies
    {(speed<=speed') && (speed=speed')>=0}
    Until $speed$ {!cruiseIncr}
)}=> { => {...} // user actions

```

In this example the temporal data variable *speed* is sampled upon the *cruiseIncr* event, and is compared to the current value (*speed'*) every cycle. The latest speed value is then saved in the pivot for next cycle's comparison.

The following example consists of a temporal averaging and min/max requirement for the cruise control system, requiring that while cruise is set and unchanged the difference between average speed and minimum speed is always less than 1% of speed.

```

TRAssert{Always ({cruiseSet}Implies
    {(n++ >=0)&& ((sum+=speed') >= 0) &&
    ((average=sum/n) >=0) &&
    ((min=(speed'<min?speed':min) >=0) &&
    (average-min < speed'/100)
    }
    Until $speed,min=1000,n=0,average=0,sum=0$
    {cruiseChange || cruiseOff}
)}=> {...} // user actions

```

In this example the only data value that is sampled from the environment (the *cruise* method/function) is *speed*. All other pivots (i.e., for *min*, *n*, *average*, and *sum*) are initialized upon the construction of the *Until* object. Likewise, the only prime value that is sampled from the environment is *speed'*, whereas all other primed variables are assigned as specified in the assignment statements (e.g. *average'=sum'/n'*). The TemporalRover makes this distinction when it recognizes an assignment in the declaration statement, such as *sum=0* above.

## 4. References

1. E. Chang, A. Pnueli, Z. Manna - *Compositional Verification of Real-Time Systems*, Proc. 9<sup>th</sup> IEEE Symp. On Logic In Computer Science, 1994, pp. 458-465.
2. D. Drusinsky - *The Temporal Rover and ATG Rover*. Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science, 1885, pp. 323-329.
3. D. Drusinsky - *Formal Specs Can Handle Exceptions*, CMP Embedded Developers Journal, Nov. 2001, pp., 10-14.
4. D. Drusinsky, *On-line Efficient Monitoring of Metric Temporal Logic Specifications using Alternating Automata*, submitted for publication.
5. D. Drusinsky and J. Fobes - *Real-time, On-line, Low Impact, Temporal Pattern Matching*, 7<sup>th</sup> World Multiconference on Systemics, Cybernetics and Informatics, Orlando FL, 2003; accepted for publication.
6. A. Pnueli - *The Temporal Logic of Programs*, Proc. 18<sup>th</sup> IEEE Symp. on Foundations of Computer Science, pp. 46-57, 1977.
7. A. P. Sistla and O. Wolfson - *Temporal Conditions and Integrity Constraints in Active Database Systems*, Proceedings of the ACM-SIGMOD 1995, International Conference on Management of Data, San Jose, CA, May 1995.

# FAST: Fast Acceleration of Symbolic Transition Systems

Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci

LSV, CNRS UMR 8643  
ENS de Cachan  
61 avenue du président Wilson  
F-94235 CACHAN Cedex  
FRANCE

`{bardin,finkel,leroux,petrucci}@lsv.ens-cachan.fr`

**Abstract.** FAST is a tool for the analysis of infinite systems. This paper describes the underlying theory, the architecture choices that have been made in the tool design. The user must provide a model to analyse, the property to check and a computation policy. Several such policies are proposed as a standard in the package, others can be added by the user. FAST capabilities are compared with those of other tools. A range of case studies from the literature has been investigated.

## 1 Introduction

Model-checking is a wide-spread technique in critical systems verification. Several efficient model-checkers, such as SMV [SMV], SPIN [SPI] or DESIGN/CPN [CPN], are available. However, these tools are restricted to finite systems whereas many real systems are infinite, because of parameters or unbounded data structures.

FAST is a tool designed to allow automatic verification of systems modeled by automata augmented with (unbounded) integer variables (*extended counter automata*). The main issue addressed by FAST is the computation of the *exact* (*infinite*) set of configurations reachable from a given set of initial configurations. Let us recall that verification of safety properties can be reduced to reachability of a given configuration from a set of initial configurations.

A lot of properties are in general undecidable, but there are two ways to deal with undecidability. The first one is to consider decidable subclasses, thus reducing the expressiveness of the model, while the second one is to accept only a semi-algorithm, which does not terminate in the general case but is expected to terminate in most practical cases. We follow the second approach. The techniques used in FAST are based on *acceleration* [FL02]. It comes down to computing the (exact) effect of iterating a control loop of *an arbitrary length (cycle)*. These cycles are automatically chosen. Both forward and backward reachability are allowed. FAST works on *linear systems*, i.e. *finite sets of linear functions* whose definition domains are defined by a *Presburger formula over non-negative integers*. Most systems with integer variables can be described in such a way. In [FL02], it is proved that for linear systems whose associated square matrices

generate a finite multiplicative monoid – namely *finite linear systems*, the acceleration of a loop terminates. It turns out that most integer variables systems appear to be finite linear systems. Even though termination is not guaranteed, in practice, FAST deals with a large amount of examples of our extended counter automata model (see section 4). We believe that Presburger arithmetic is sufficient to model these problems and that most systems are effectively computable.

FAST is freely available on Jérôme Leroux home page [FAS].

## 2 Related Tools

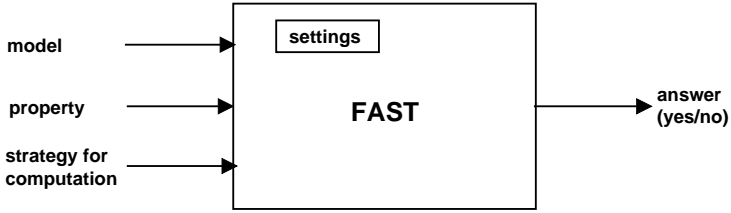
Table 1 summarises a comparison of the main tools able to cope with integer variables infinite systems. No performance comparison is provided, as these tools do not require the same user implication, they take different input models, more or less limited, and hence perform different computations.

**Table 1.** A comparison of different tools for reachability set computation

	variable type	guards	actions	acceleration	auto. cycle search	forward	backward	exact comput.	engine
FAST	$\mathbb{N}$	Presburger	$\vec{x} = A.\vec{x} + \vec{b}$	yes	yes	yes	yes	yes	LNDD, MONA
LASH	$\mathbb{Z}$	convex sets	$\vec{x} = A.\vec{x} + \vec{b}$	yes	no	yes	yes	yes	NDD
	$\mathbb{R}$	convex sets	$\vec{x} = A.\vec{x} + \vec{b}$	no	-	yes	yes	yes	RVA
Trex	$\mathbb{Z}$	$\bigwedge \begin{cases} x_i \leq x_j + c \\ x_i \leq c \end{cases}$	$\bigwedge \begin{cases} x_i = x_j + c \\ x_i = c \end{cases}$	yes	yes	yes	yes	yes	PDBM
	$\mathbb{R}$	$\bigwedge \begin{cases} x_i \geq c \end{cases}$	$\bigwedge \begin{cases} x_i = x_j \\ x_i = 0 \end{cases}$	yes	yes	yes	yes	no	PDBM
BRAIN	$\mathbb{N}$	$\bigwedge x_i \leq x_j + c$	$\vec{x} = A.\vec{x} + \vec{b}$	no	-	no	yes	yes	period basis
BABYLON	$\mathbb{N}$	$\bigwedge x_i \leq x_j + c$	$\vec{x} = A.\vec{x} + \vec{b}$	no	-	no	yes	no	CST
HYTECH	$\mathbb{R}$	convex sets	$\vec{x} = A.\vec{x} + \vec{b}$	no	-	yes	yes	no	convex polyhedra
ALV	$\mathbb{Z}$	Presburger	Presburger	no	-	yes	yes	yes	OMEGA, BDD

## 3 Architecture

Figure 1 shows the inputs and outputs of FAST. FAST requires a model of the system to analyze, a reachability property to check and a strategy to direct the computation. If it terminates, the tool answers whether the property is satisfied or not. *Settings* can also be optionally set by the user, such as the ordering of variables and stop criteria.



**Fig. 1.** FAST inputs and outputs

*Strategies* allow the user to direct “by-hand” the computation. Strategies make it possible to describe standard model-checking features such as forward or backward reachability as well as more advanced constructs like a sequence of incremental submodel analysis. This has been successfully used to verify the TTP protocol (see section 4). Concretely, the user describes strategies through a high-level language allowing to manipulate Presburger definable sets of integers, linear functions, booleans and providing primitives for *pre\** and *post\** operations.

Presburger definable sets of integers are internally represented by Labeled Number Decision Diagrams (LNDDs). LNDDs allow to represent any Presburger formula and provides basic operations on sets (intersection, negation, inclusion or emptiness test) as well as more advanced constructs like the acceleration of a cycle described in [FL02]. Our implementation uses packages from MONA [MON], providing automata operations. An extended version of FAST for integer arithmetic has also been developed, built on LASH libraries. But there was a drop in performances, integer variables being implicitly encoded as two non-negative integer variables. Since all the case studies considered only deal with non-negative integers, we decided to first limit FAST to non-negative integers.

## 4 Results

FAST has been applied to a large number of examples (about 40), ranging from Petri nets to abstract multi-threaded JAVA programs, mainly taken from [Del]. About 80% of these case studies could effectively be verified. It proves that choices made during FAST design, like having only a semi-algorithm or restricting FAST to non-negative integers, are sound for practical infinite systems verification. Moreover, most of these examples require only a basic predefined strategy (a forward search), thus only little input from the user.

Figure 2 presents the performances obtained by FAST on ten of the most representative examples. Dekker ME is a bounded Petri net, other examples are infinite state systems because of parameters (lift controller) or unbounded integer variables (FMS). Despite its number of variables and transitions, the Swimming Pool protocol is a highly non-trivial protocol. The TTP protocol is a complex group membership protocol, using elaborate guards. The tool computes efficiently these examples. A forward search has been used for all examples. For the particular case of TTP, a more elaborate strategy was also tested, leading to considerable increase in computation time.

Case Study	variables	transitions	time (s)	mem.(MB)	n. states	n. acc	c. length	n. cycles
Dekker ME	22	22	21.72	5.48	5	36	1	22
CSM	13	13	45.57	6.31	6	32	2	35
FMS	22	20	157.48	8.02	21	23	2	46
Swimming Pool	9	6	111	29.06	30	9	4	47
Producer/Consumer with Java threads - N	18	14	723.27	12.46	58	86	2	75
Lift Controller - N	4	5	4.56	2.90	14	4	3	20
TTP	10	17	1186.24	73.24	1140	31	1	17
TTP (ad hoc strategy)	10	17	246.67	72.87	1140	16	1	17

**Fig. 2.** Results using an Intel Pentium 933 MHz with 512 Mbytes

Considering the case studies that could not be verified (9 out of 40), we propose three reasons for FAST not to terminate. First of all, the input model can be such that FAST cannot terminate, either some loops have infinite associated monoids or the reachability set is not *flatable*, i.e. not computable using a finite set of accelerations [FL02]. Second, the computation may lead to large automata and saturate the memory. Finally, there may be too many cycles to consider, and then the heuristic used by FAST to find cycles to be accelerated reaches its limits.

We currently develop an interface, for the user to set parameters, e.g. the cycle length, which will allow us to cope with these drawbacks.

## References

- [ALV] ALV homepage. <http://www.cs.ucsb.edu/~bultan/composite/>.
- [BRA] The BRAIN homepage. <http://www.cs.man.ac.uk/~voronkov/BRAIN/>.
- [CPN] DESIGN/CPN online. <http://www.daimi.au.dk/designCPN>.
- [Del] G. Delzanno. Home Page – Giorgio Delzanno. <http://www.disi.unige.it/person/DelzannoG/>.
- [FAS] FAST homepage. <http://www.lsv.ens-cachan.fr/~leroux/fast/>.
- [FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002)*, Kanpur, India, Dec. 2002, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.
- [HYT] HYTECH homepage. <http://www-cad.eecs.berkeley.edu/~tah/HyTech/>.
- [LAS] The LASH Toolset. <http://www.montefiore.ulg.ac.be/~boigelot/research/lash/>.
- [MON] the MONA project. <http://www.brics.dk/mona/>.
- [SMV] SMV homepage. <http://www-cad.eecs.berkeley.edu/~kenmcml/>.
- [SPI] SPIN homepage. <http://spinroot.com/spin/>.
- [TRE] TREX homepage. <http://www.liafa.jussieu.fr/~sighirea/trex/>.

# Rabbit: A Tool for BDD-Based Verification of Real-Time Systems

Dirk Beyer, Claus Lewerentz, and Andreas Noack

Software Systems Engineering Research Group  
Brandenburg Technical University at Cottbus, Germany  
{db|cl|an}@informatik.tu-cottbus.de

**Abstract.** This paper gives a short overview of a model checking tool for real-time systems. The modeling language are timed automata extended with concepts for modular modeling. The tool provides reachability analysis and refinement checking, both implemented using the data structure BDD. Good variable orderings for the BDDs are computed from the modular structure of the model and an estimate of the BDD size. This leads to a significant performance improvement compared to the tool RED and the BDD-based version of Kronos.

## 1 Introduction

*Timed automata* are a common and theoretically well-founded formalism for real-time systems [1]. Reachability analysis of timed automata has been implemented in several tools, the best-known being Kronos [11] and Uppaal [2]. From our point of view, two major problems are the lack of concepts for modeling large systems and the exploding consumption of time and memory by the verification algorithms. We address these issues with our tool *Rabbit*, which provides the following features:

- **Modular Modeling.** Our modular extension of timed automata is called Cottbus Timed Automata (CTA). The automata are encapsulated by modules. Each module has an explicit *interface*, which declares the variables and synchronization labels used for the communication with other modules. The use of these variables and synchronization labels can be restricted by specifying an explicit access mode (read only, exclusive write, etc.). Replicated subsystems do not have to be multiply defined, but can be instantiated from a common *template module*. Modules can contain other modules to form hierarchical structures. Our formalism provides a *compositional semantics*, i.e. we can define the semantics of a CTA module on the basis of the semantics of its components [7].
- **Reachability Analysis.** The tool provides efficient reachability analysis for timed automata. Sets of configurations are represented by the data structure binary decision diagram (BDD). The modular structure of the model is used to compute BDD variable orderings for an efficient representation of the transition relation and the set of reachable configurations.
- **Refinement Checking.** To make the verification of large systems tractable, detailed modules can be replaced by more abstract modules. To prove the correctness of

such a replacement regarding safety properties, we have to check that two modules have the same behavior with respect to external synchronization labels, i.e. the set of traces of the abstract module has to be a superset of the set of traces of the detailed module. To enable efficient modular proofs, we check the existence of a *simulation relation* in our tool implementation. A simulation relation exists in many practical cases of trace inclusion, especially when stepwise refinement is used in the development process. Details about the implementation of this refinement check are given in [4].

For a comprehensive and detailed explanation of all the concepts and the tool, we refer to [5]. The tool Rabbit, example models and related papers are available from <http://www.software-systemtechnik.de/Rabbit>.

## 2 BDD-Based Reachability Analysis

Safety properties of timed automata can be verified by reachability analysis. The main problem is the exploding consumption of time for the computation and memory for the representation of the reachable configurations. Therefore the data structure for sets of configurations is of vital importance. Sets of configurations of timed automata consist of locations and associated sets of clock assignments. For the symbolic representation of sets of locations binary decision diagrams (BDDs) are widely used. For a uniform representation of locations and clock assignments as BDDs we defined an **integer semantics** which only considers integer clock assignments. We proved that for timed automata without strict clock constraints (i.e. without  $<$  and  $>$  in clock constraints), this integer semantics is equivalent to the usual, continuous semantics with respect to the reachable locations [3]. The restriction to non-strict clock constraints is of technical nature, and we did not find examples within our application area of production cell controllers and real-time algorithms for which it is difficult to construct models without strict constraints.

Representing the transition relation as several BDDs and applying these **partial transition relations** sequentially is more efficient than using a monolithic transition relation [9]. Concerning the **order of their application**, always computing the fixed point of the discrete transitions before applying time transitions is a successful strategy to avoid large intermediate BDDs.

We use a heuristic to find good BDD **variable orderings**. This heuristic first computes an initial variable ordering and then improves it by local search. The initial variable ordering is a pre-order linearization of the module hierarchy (i.e. the modules are in the order in which they are reached by a depth-first traversal of the hierarchy). This assigns local components of a module to neighboring positions in the variable ordering, and thus exploits the knowledge of the modeler who usually tries to create cohesive modules. Then local search is applied to improve the ordering with respect to a size estimate for the BDD of the reachability set [3]. This estimate reflects the two most important characteristics for good variable orderings: (1) Communicating components have neighboring positions within the ordering. (2) Components which communicate with many other components precede these other components.

**Table 1.** Time for the computation of the reachability set of Fischer’s protocol

# proc.	4	5	6	7	8	10	12	14	16	32	64	128
Uppaal	0.06	1.44	181	32488								
RED	1.64	6.78	21.7	60.7	168	1400						
Rabbit	0.04	0.08	0.15	0.26	0.50	1.35	1.61	3.81	6.50	61.4	559	5200

**Table 2.** Time for the computation of the reachability set of the FDDI protocol

# senders	2	4	6	8	10	12	14	16
Uppaal	0.01	0.03	0.16	1.42	18.2	279	4530	
RED	0.02	0.09	0.26	0.61	1.18	2.16	3.62	6.31
Rabbit	0.04	0.25	0.99	4.20	11.4	26.9	49.8	142

### 3 Performance Results

Performance results are given in seconds of CPU time on a Linux PC with an AMD Athlon processor (1 GHz) for the publicly available tools RED [10] version 3.1, which is based on a BDD-like data structure called CRD, Rabbit [4] version 2.1, which is based on BDDs with automatic variable ordering, and Uppaal2k [2] version 3.2.4 (without approximation), a popular and highly optimized DBM-based tool. An empty table entry means that the analysis needs more than 400 MB memory or more than 7 200 s computation time.

Detailed analytical explanations of the experimental results can be found in [6]. This paper also discusses the sensitivity of the BDD representation to the size of constants in the model, which is a major disadvantage of BDDs compared to CRDs and DBMs.

**Fischer’s Protocol** (Table 1). Fischer’s timing-based protocol for mutual exclusion is a protocol for accessing a shared resource. We computed the set of reachable configurations to verify the mutual exclusion property. For the BDD-based version of Kronos (which is not publicly available) a maximum of 14 processes is reported in [8].

The example of Fischer’s protocol illustrates the dramatic influence of the variable ordering: The set of all reachable configurations for 16 processes is represented by 2 096 957 BDD nodes using an ordering with the shared variable  $k$  at the last position. This ordering violates the second characteristic for good orderings because variable  $k$  is used by all processes. A good variable ordering with variable  $k$  at the first position reduces the representation to 6 190 BDD nodes.

**Token-Ring-FDDI Protocol** (Table 2). Fiber Distributed Data Interface (FDDI) is a high speed protocol for local networks based on token ring technology. The automata model was introduced by Yovine [12]. Here RED outperforms Rabbit because the number of reachable locations does not explode with growing number of senders.

**CSMA/CD Protocol** (Table 3). CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) is a protocol for communication on a broadcast network with a multiple access medium. The timed automata model is taken from [11].

**Production Cell.** To validate the suitability of our tool for more realistic models, we developed a CTA model of a production cell. This system consists of 20 machines and belts with 44 sensors and 28 motors. We modeled the system as a modular composi-



**Table 3.** Time for the computation of the reachability set of the CSMA/CD protocol

# senders	2	4	6	8	10	12	14	16	32	64	128	256
Uppaal	0.01	0.03	5.1									
RED	0.05	0.28	1.15	5.88	41.4	516						
Rabbit	0.02	0.08	0.23	0.49	0.82	1.28	1.83	2.69	12.6	62.9	367	2160

tion of several belts, turntables and machines, using 82 timed automata with 44 clocks, 17 discrete variables and 183 synchronization labels. For a model with a simple communication structure like Fischer's protocol good variable orderings might be obvious (at least for experts). The production cell shows the need for automatic variable ordering, and the effectiveness of our automatic estimate-based heuristic: The pre-order linearization of the module hierarchy results in a BDD for the reachability set with 378 229 nodes. This is a good ordering, much better than the vast majority of other permutations, but applying our heuristic improves the representation to 14 895 nodes. Modular proofs using refinement checking simplify the models used in reachability analysis and thus further reduce the BDD size.

## References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannot, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In *Proc. MOVEP'00*, LNCS 2067, pages 99–124. Springer, 2001.
3. Dirk Beyer. Improvements in BDD-based Reachability Analysis of Timed Automata. In *Proc. FME'01*, LNCS 2021, pages 318–343. Springer, 2001.
4. Dirk Beyer. Efficient Reachability Analysis and Refinement Checking of Timed Automata using BDDs. In *Proc. CHARME'01*, LNCS 2144, pages 86–91. Springer, 2001.
5. Dirk Beyer. *Formale Verifikation von Realzeit-Systemen mittels Cottbus Timed Automata*. Verlag Mensch & Buch, Berlin, 2002. Zugl.: Dissertation, BTU Cottbus, 2002.
6. Dirk Beyer and Andreas Noack. A comparative study of decision diagrams for real-time verification. Technical Report I-03/2003, BTU Cottbus, 2003.
7. Dirk Beyer and Heinrich Rust. Cottbus Timed Automata: Formal Definition and Semantics. In *Proc. FSCBS'01*, pages 75–87, 2001.
8. Marius Bozga, Oded Maler, Amir Pnueli, and Sergio Yovine. Some progress on the symbolic verification of timed automata. In *Proc. CAV'97*, LNCS 1254, pages 179–190, 1997.
9. Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on CAD*, 13(4):401–424, April 1994.
10. Farn Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In *Proc. FORTE'01*, pages 235–250. Kluwer, 2001.
11. Sergio Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1-2):123–133, October 1997.
12. Sergio Yovine. Model checking timed automata. In *Lectures on Embedded Systems*, LNCS 1494, pages 114–152. Springer, 1998.

# Making Predicate Abstraction Efficient: How to Eliminate Redundant Predicates\*

Edmund Clarke<sup>1</sup>, Orna Grumberg<sup>2</sup>, Muralidhar Talupur<sup>1</sup>, and Dong Wang<sup>1</sup>

<sup>1</sup> Carnegie Mellon University

<sup>2</sup> TECHNION - Israel Institute of Technology

**Abstract.** In this paper we consider techniques to identify and remove redundant predicates during predicate abstraction. We give three criteria for identifying redundancy. A predicate is redundant if any of the following three holds (i) the predicate is equivalent to a propositional function of other predicates. (ii) removing the predicate preserves safety properties satisfied by the abstract model (iii) removing it preserves bisimulation equivalence. We also show how to efficiently remove the redundant predicates once they are identified. Experimental results are included to demonstrate the effectiveness of our methods.

**Keywords:** predicate abstraction, redundancy, simulation, bisimulation, safety properties

## 1 Introduction

Abstraction has been widely accepted as a viable way for reducing the complexity of systems during temporal logic model checking [10]. Predicate abstraction [1,2,3,11,12,14,19,21,22] has emerged as one of the most successful abstraction techniques. It has been used in both software and hardware verification. In this paper, we give a technique to improve predicate abstraction by eliminating redundant predicates. This technique can be applied in both software and hardware verification. We give efficient verification algorithms for finite state systems (hardware) and outline how our method can be used for infinite state systems (software).

In predicate abstraction, the number of predicates affects the overall performance. Since each predicate corresponds to a boolean state variable in the abstract model, the number of predicates directly determines the complexity of building and checking the abstract model. Most predicate abstraction systems build an abstract model of the system to be verified. While building the abstract model, the number of calls made to a theorem prover (or a SAT solver in our case) can be exponential in the number of predicates. Consequently, it is desirable to use as few predicates as possible. Existing techniques for choosing relevant predicates may use more predicates than necessary to verify a given

---

\* This research is sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the Gigascale Silicon Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9803774. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, GSRC, NSF, or the United States Government.

property. That is some of the predicates used can be redundant (the precise definition of redundancy is given later).

*Counterexample guided abstraction refinement* (CEGAR) [7,16,20] is an example of a commonly used abstraction technique. It works by introducing new predicates to eliminate spurious counterexamples. The new predicates depend on certain abstract states in the spurious abstract counterexample. Thus, different predicates are likely to be closely related when similar abstract counterexamples occur and this might lead to redundancy in the predicate set. These similarities may result in the following two cases: (a) A predicate may be logically equivalent to a propositional formula in terms of other predicates. (b) For the predicate  $P$  under consideration, there exist two nontrivial propositional formulas  $P_{sub}$  and  $P_{sup}$  in terms of other predicates such that  $P_{sub}$  implies  $P$  and  $P$  implies  $P_{sup}$ . It is obvious that when case (a) happens, the predicate is redundant. This predicate can be replaced by the equivalent formula and we thus obtain a new abstract model. We call the original abstract model the *current/original abstract model* and the new one the *reduced abstract model*. It is easy to show that the two models are bisimilar. In the other case, a predicate  $P$  satisfying case (b) may not be redundant. More conditions on the abstract model are needed to ensure that replacing  $P$  by  $P_{sub}$  or  $P_{sup}$  will not affect the results of model checking the abstract model. We have identified two redundancy conditions for case (b), one that preserves safety properties (that is the original and the reduced abstract models both satisfy the same safety properties) and one that preserves bisimulation equivalence (that is the original and the reduced abstract models are bisimulation equivalent). different situations and there are cases where one works better than the other. Altogether there are three different redundancy conditions. One useful feature of our redundancy conditions is that they do not require exact computation, we can use approximations and still identify redundancy.

Removing a predicate involves constructing the abstract model using the reduced predicate set. We give a simple method to construct the reduced abstract model from the original abstract model in Section 4.

## 1.1 Related Work

The notion of redundancy has been explored in resolution theorem proving [5], where it is called *subsumption*. Intuitively a clause is considered redundant if it is logically implied via substitution by other clauses. Our conditions for redundancy are more complicated. Even if a predicate is implied by other predicates, we still need to consider the abstract transition relation in order to decide whether removing the predicate will affect the results of verifying a given property.

The work that is closest to ours is the notion of *strengthening* in [1]. To build the abstract model, the weakest precondition is converted to an expression over the set of predicates in the abstraction. Thus strengthening is somewhat similar to the *replacement function* in this paper. However, in [1], the result of the strengthening is over all the predicates, while the replacement function used here is defined over a subset of the predicates. Finally, the two transformations have different purposes. Strengthening is only used to build an abstract model; while our transformation is used to remove redundant predicates and thus reduce the complexity of the abstract model.

This paper removes unnecessary predicates introduced by counterexample guided refinement. Recently, abstraction refinement with more than one counterexamples has been investigated in [18,13]. However, there is no guarantee for the elimination of redundant predicates by considering multiple counterexamples in computing predicates alone. Thus, our techniques can also be applied in that context.

Exploiting functional dependencies between state variables to reduce BDD size has been investigated in [15]. In that approach, if a variable can be shown to be a function of other variables, it can be eliminated during BDD based verification. Our approach is more general in that it is possible to remove a predicate even if it is not equivalent to any function over other predicates.

## 1.2 Outline of the Paper

In the next section we introduce predicate abstraction and other relevant theory. In Section 3 we define the *replacement function* and show how to compute it. In the next section we show how to construct the new abstract model after removing a redundant predicate. In Section 5, the simplest form of redundancy, called equivalence induced redundancy, is presented. Section 6 and Section 7 give redundancy conditions that preserve safety properties and bisimulation equivalence respectively. The comparison between redundancies in the last two sections is illustrated using examples in Section 8. We discuss how our algorithms can be applied to software verification in Section 9. In Section 10, we describe our experiments. Section 11 concludes the paper with some directions for future research.

## 2 Preliminaries

In this section we review the relevant theory of property preserving abstractions introduced by Clarke, Grumberg and Long in [6] and Loiseaux et al. in [17]. Using this theory we describe the predicate abstraction framework of Saidi and Shankar [22].

### 2.1 Notation

Let  $S_1$  and  $S_2$  be sets of states, and let  $f$  be a function mapping the powerset of  $S_1$  to the powerset of  $S_2$ , i.e.,  $f : 2^{S_1} \rightarrow 2^{S_2}$ . The *dual of the function*  $f$  is defined to be

$$\tilde{f}(X) = \overline{f(\overline{X})},$$

where the overbar indicates complementation in the appropriate set of states.

Let  $\rho$  be a relation from  $S_1$  to  $S_2$ , and let  $A$  be a subset of  $S_2$ , then the function  $pre[\rho](A)$  gives the *preimage* of  $A$  under the relation  $\rho$ . Formally,

$$pre[\rho](A) = \{s_1 \in S_1 \mid \exists s_2 \in A. \rho(s_1, s_2)\}.$$

Similarly, let  $B$  be a subset of  $S_1$ , then the function  $post[\rho](B)$  gives the *postimage* of  $B$  under the relation  $\rho$ . More formally,

$$post[\rho](B) = \{s_2 \in S_2 \mid \exists s_1 \in B. \rho(s_1, s_2)\}$$

If relation  $\rho$  is a total function on  $S_1$ , then  $\widetilde{pre}[\rho]$  is the same as  $pre[\rho]$  [17].

We will be reasoning about a concrete state machine and an abstraction of that machine. For establishing a relationship between the set of concrete states  $S_1$  and the set of abstract states  $S_2$  we will use the concept of a *Galois connection*.

**Definition 1.** Let  $Id_S$  denotes the identity function on the powerset of  $S$ . A Galois connection between  $2^{S_1}$  and  $2^{S_2}$  is a pair of monotonic functions  $(\alpha, \gamma)$ , where  $\alpha : 2^{S_1} \rightarrow 2^{S_2}$  and  $\gamma : 2^{S_2} \rightarrow 2^{S_1}$ , such that  $Id_{S_1} \subseteq \gamma \circ \alpha$  and  $\alpha \circ \gamma \subseteq Id_{S_2}$ .

The functions  $\alpha$  and  $\gamma$  are often called the *abstraction function* and the *concretization function*, respectively. The Galois connection that we will be using in this paper is described in the following proposition.

**Proposition 1.** [17] Given a relation  $\rho \subseteq S_1 \times S_2$ , the pair  $(post[\rho], \widetilde{pre}[\rho])$  is a Galois connection between  $2^{S_1}$  and  $2^{S_2}$ .

We denote this Galois connection by  $(\alpha_\rho, \gamma_\rho)$ . Sometimes, we write  $(\alpha, \gamma)$  when the relation  $\rho$  is clear from the context.

## 2.2 Existential Abstraction

We model circuits and programs as transition systems. Given a set of atomic propositions,  $A$ , let  $M = (S, S_0, R, L)$  be a *transition system* (refer to [9] for details).

**Definition 2.** Given two transition systems  $M = (S, S_0, R, L)$  and  $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ , with atomic propositions  $A$  and  $\hat{A}$  respectively, a relation  $\rho \subseteq S \times \hat{S}$ , which is total on  $S$ , is a *simulation relation* between  $M$  and  $\hat{M}$  if and only if for all  $(s, \hat{s}) \in \rho$  the following conditions hold:

- $L(s) \cap \hat{A} = \hat{L}(\hat{s}) \cap A$
- For each state  $s_1$  such that  $(s, s_1) \in R$ , there exists a state  $\hat{s}_1 \in \hat{S}$  with the property that  $(\hat{s}, \hat{s}_1) \in \hat{R}$  and  $(s_1, \hat{s}_1) \in \rho$ .

We say that  $\hat{M}$  *simulates*  $M$  through the simulation relation  $\rho$ , denoted by  $M \preceq_\rho \hat{M}$ , if for every initial state  $s_0$  in  $M$  there is an initial state  $\hat{s}_0$  in  $\hat{M}$  such that  $(s_0, \hat{s}_0) \in \rho$ . We say that  $\rho$  is a *bisimulation relation* between  $M$  and  $\hat{M}$  if  $M \preceq_\rho \hat{M}$  and  $\hat{M} \preceq_{\rho^{-1}} M$ . If there is a bisimulation relation between  $M$  and  $\hat{M}$  then we say that  $M$  and  $\hat{M}$  are *bisimilar*, and we denote this by  $M \equiv_{bis} \hat{M}$ .

**Theorem 1.** (Preservation of ACTL\* [9])

Let  $M = (S, S_0, R, L)$  and  $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$  be two transition systems, with  $A$  and  $\hat{A}$  as the respective sets of atomic propositions and let  $\rho \subseteq S \times \hat{S}$  be a relation such that  $M \preceq_\rho \hat{M}$ . Then, for any ACTL\* formula,  $\Phi$  with atomic propositions in  $A \cap \hat{A}$

$$\hat{M} \models \Phi \text{ implies } M \models \Phi.$$

In the above theorem, if  $\rho$  is a bisimulation relation, then for any CTL\* formula  $\Phi$  with atomic propositions in  $A \cap \hat{A}$ ,  $\hat{M} \models \Phi \Leftrightarrow M \models \Phi$ .

Let  $M = (S, S_0, R, L)$  be a concrete transition system over a set of atomic propositions  $A$ . Let  $\hat{S}$  be a set of abstract states and  $\rho \subseteq S \times \hat{S}$  be a total function on  $S$ . Further, let  $\rho$  and  $L$  be such that for any  $\hat{s} \in \hat{S}$ , all states in  $pre[\rho](\hat{s})$  have the same labeling over a subset  $\hat{A}$  of  $A$ . Then an abstract transition system  $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$  over  $\hat{A}$  which simulates  $M$  can be constructed as follows:

$$\hat{S}_0 = post[\rho](S_0) = \exists s. S_0(s) \wedge \rho(s, \hat{s}) \quad (1)$$

$$\hat{R}(\hat{s}, \hat{s}') = \exists s s'. \rho(s, \hat{s}) \wedge \rho(s', \hat{s}') \wedge R(s, s') \quad (2)$$

$$\text{for each } \hat{s} \in \hat{S}, \hat{L}(\hat{s}) = \bigcap_{s \in pre[\rho](\hat{s})} (L(s) \cap \hat{A}) \quad (3)$$

**Proposition 2.** For  $M$  and  $\hat{M}$  in the above construction  $M \preceq_\rho \hat{M}$

In the above construction  $\hat{R}$  is defined in terms of the abstract current state  $\hat{s}$  and the abstract next state  $\hat{s}'$ . This construction is from [6], and it is also implicit in the paper by Loiseaux et al. [17]. This kind of abstraction is called *existential abstraction*. The set of initial states in the abstract system are those states of  $\hat{S}$  that are related to the initial states of  $M$ . Note that for any two states  $s$  and  $\hat{s}$  related under  $\rho$  the property  $L(s) \cap \hat{A} = \hat{L}(\hat{s})$  holds.

### 2.3 Predicate Abstraction

Predicate abstraction can be viewed as a special case of existential abstraction. In predicate abstraction a set of predicates  $\{P_1, \dots, P_k\}$ , including those in the property to be verified, are identified from the concrete program. These predicates are defined on the variables of the concrete system. They also serve as the atomic propositions that label the states in the concrete and abstract transition systems. That is, the set of atomic propositions is  $A = \{P_1, P_2, \dots, P_k\}$ . A state in the concrete system will be labeled with all the predicates it satisfies. The abstract state space has a boolean variable  $B_j$  corresponding to each predicate  $P_j$ . So each abstract state is a valuation of these  $k$  boolean variables. An abstract state will be labeled with predicate  $P_j$  if the corresponding bit  $B_j$  is 1 in that state. The predicates are also used to define a total function  $\rho$  between the concrete and abstract state spaces. A concrete state  $s$  will be related to an abstract state  $\hat{s}$  through  $\rho$  if and only if the truth value of each predicate on  $s$  equals the value of the corresponding boolean variable in the abstract state  $\hat{s}$ . Formally,

$$\rho(s, \hat{s}) = \bigwedge_{1 \leq j \leq k} P_j(s) \Leftrightarrow B_j(\hat{s}) \quad (4)$$

Note that  $\rho$  is a total function because each  $P_j$  can have one and only one value on a given concrete state and so the abstract state corresponding to the concrete state is unique. Based on Section 2.1, the pair of functions  $post[\rho]$  and  $\widetilde{pre}[\rho]$  generated from relation  $\rho$  forms a Galois connection. We will denote this Galois connection by  $(\alpha, \gamma)$ .

Note that since  $\rho$  is a total function,  $\widehat{pre}[\rho] = pre[\rho]$ . Using this  $\rho$  and the construction given in the previous subsection, we can build an abstract model which simulates the concrete model. In [22] the abstract transition relation  $\hat{R}$  is defined as

$$\bigwedge \{ \hat{Y} \rightarrow \hat{Y}' \mid \forall V, V'. ((R(V, V') \wedge \gamma(\hat{Y})) \rightarrow \gamma(\hat{Y}')) \} \quad (5)$$

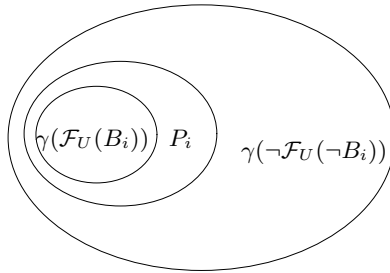
where  $\hat{Y}$  is an arbitrary conjunction of the literals of the current state variables  $\{B_1, B_2, \dots, B_k\}$  and  $\hat{Y}'$  is an arbitrary disjunction of literals of the next state variables  $\{B'_1, B'_2, \dots, B'_k\}$ . It can be shown that (5) is equivalent to (2).

### 3 The Replacement Function

Our goal is to eliminate  $B_i$  from the abstract model  $\hat{M}$  without sacrificing accuracy. For this purpose, we define an under-approximation,  $\mathcal{F}_U(B_i)$ , for  $B_i$  in terms of the other variables. More precisely, let  $M$  be a concrete transition system,  $\{P_1, P_2, \dots, P_k\}$  be a set of predicates defined on the states of  $M$ , and let  $\rho$  be a total function defined by equation (4). Also, let  $\hat{M}$  be the corresponding abstract transition system over  $V = \{B_1, B_2, \dots, B_k\}$ . The support of an abstract set of states  $\hat{S}_1$  includes  $B_i$  if and only if

$$\exists \hat{s} \in \hat{S}_1. \hat{s}[B_i \leftarrow 0] \in \hat{S}_1 \not\Rightarrow \hat{s}[B_i \leftarrow 1] \in \hat{S}_1.$$

Consider the boolean variable  $B_i$  and the set  $U = V \setminus \{B_i\}$ . Let  $\Phi$  denote either  $B_i$  or  $\neg B_i$ . The *replacement function* for  $\Phi$ , denoted by  $\mathcal{F}_U(\Phi)$ , is defined as the largest set of *consistent* abstract states (we call an abstract state *consistent* if its concretization is not empty) whose support is included in  $U$  and whose concretization is a subset of  $\gamma(\Phi)$ . The implications  $\gamma(\mathcal{F}_U(B_i)) \rightarrow \gamma(B_i)$  and  $\gamma(\mathcal{F}_U(\neg B_i)) \rightarrow \gamma(\neg B_i)$  follow from this definition. Figure 1 shows the relationship between the concretization of a predicate  $B_i$ ,  $\mathcal{F}_U(B_i)$ , and  $\neg \mathcal{F}_U(\neg B_i)$ .



**Fig. 1.** Relationship between the concretization of  $B_i$ ,  $\mathcal{F}_U(B_i)$ , and  $\neg \mathcal{F}_U(\neg B_i)$

We now show how to compute  $\mathcal{F}_U(B_i)$ . Consider the abstract state space  $\hat{S}$  given by tuples of the form  $(B_1, B_2, \dots, B_k)$ . Not all the abstract states have corresponding concrete states. We consider only the set of consistent abstract states,  $g$ , that are related

to some concrete states by the relation (4) in Section 2.3. Formally, if  $S$  is the set of concrete states,  $\{P_j | 1 \leq j \leq k\}$  is the set of predicates and  $\rho$  is the simulation relation as in Section 2.3, then

$$g = \text{post}[\rho](\text{true}) = \{\hat{s} \mid \exists s \in S. \bigwedge_{1 \leq j \leq k} P_j(s) \Leftrightarrow B_j(\hat{s})\}.$$

For hardware verification, all the concrete state variables have finite domain. The set  $g$  can be efficiently computed using OBDDs [4] through a series of conjunction and quantification operations. However, a general theorem prover is needed for infinite state systems. We define  $g|_{B_i}$  to be the set of reduced abstract states obtained by taking all the states in  $g$  that have the bit  $B_i$  equal to 1 and dropping the bit  $B_i$ . Similarly  $g|_{\neg B_i}$  is obtained by taking all those states in  $g$  with bit  $B_i$  equal to 0 and dropping the bit  $B_i$ . The following theorem shows that the set  $(g|_{B_i} \wedge \neg g|_{\neg B_i})$  is a candidate for  $\mathcal{F}_U(B_i)$ .

**Theorem 2.** *Let  $V = \{B_1, B_2, \dots, B_k\}$  be the boolean variables. Let  $U = V \setminus \{B_i\}$ , and let  $f_1 = g|_{B_i} \wedge \neg g|_{\neg B_i}$  be a set of abstract states. Then  $\gamma(f_1) \Rightarrow \gamma(B_i)$  and  $f_1$  is the largest set of consistent abstract states that does not have bit  $B_i$  in its support. Likewise, if  $f_2 = g|_{\neg B_i} \wedge \neg g|_{B_i}$ , then  $\gamma(f_2) \Rightarrow \gamma(\neg B_i)$  and  $f_2$  is the largest set of consistent abstract states that does not have bit  $B_i$  in its support.*

Replacement function is used extensively in the later sections. The correctness of our algorithms only depend on the property that  $\gamma(\mathcal{F}_U(B_i)) \rightarrow \gamma(B_i)$ . The nice advantage of this is we can use any  $f$  that satisfies  $\gamma(f) \rightarrow \gamma(B_i)$  instead of using  $\mathcal{F}_U(B_i) = f_1$ , which is difficult to compute when there are many predicates. Instead we use the following approximation: we first partition predicates into clusters as in Section 5, then compute the set of consistent abstract states and replacement function for each cluster separately. We use these easy to compute approximations to identify and remove redundant predicates. This does not affect the correctness (i.e., every identified predicate is indeed redundant), but some redundant predicates may fail to be identified.

## 4 Removing Redundant Predicates

Removal of a predicate involves constructing a new abstract transition system from the old abstract transition system. The state space of the new abstract transition system is the set of all possible valuations of the boolean variables corresponding to the new predicate set. The new predicate set has one less predicate than the old predicate set. Let  $P_i$  be the redundant predicate that is to be removed. If the old state space is given by  $k$ -tuples  $(B_1, B_2, \dots, B_k)$ , then the new state space is given by  $(k-1)$ -tuples  $(B_1, \dots, B_{i-1}, B_{i+1}, \dots, B_k)$ . Suppose the original abstract model is  $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$ . We now describe how to construct the new abstract model,  $M_r = (S_r, S_{0,r}, R_r, L_r)$  ( $r$  for “reduced”), from  $\hat{M}$  if we decide to drop the predicate  $P_i$ . The relation  $\rho_r$  between the concrete state space and the reduced state space is

$$\rho_r(s, s_r) = \bigwedge_{1 \leq j \leq k \wedge j \neq i} P_j(s) \Leftrightarrow B_j(s_r).$$



The construction of the new state space is straightforward: we just drop the boolean variable  $B_i$ . The labeling  $L_r$  is as described in Section 2.3: a reduced abstract state  $s_r$  is labeled with a predicate  $P_j$  if and only if the corresponding bit  $B_j$  is 1 in that state. The new transition relation  $R_r$  is obtained from the original abstract transition relation  $\hat{R}$  by the following equation

$$R_r(s_r, s'_r) = \exists b_i, b'_i. \hat{R}(\langle s_r, b_i \rangle, \langle s'_r, b'_i \rangle) \quad (6)$$

where  $\langle s_r, b_i \rangle$  stands for the state (in the original abstract model) obtained by inserting  $b_i$  into  $s_r$  as the  $i$ -th bit. Thus two reduced abstract states are related if there are two related states in the original abstract model that are “extensions” of these reduced abstract states. The reduced initial set of states can be similarly constructed using existential quantification as follows

$$S_{0r}(s_r) = \exists b_i. \hat{S}_0(\langle s_r, b_i \rangle) \quad (7)$$

**Lemma 1.** *The transition relation of the reduced abstract model defined by equation (6) is the same as the one built directly from the concrete model using equation (2) and  $\rho_r$  over the reduced set of predicates.*

Thus,  $R_r$  constructed using equation (6) is equivalent to the one constructed directly from the concrete model using equation (2).

## 5 Equivalence Induced Redundancy

In this section, we present the simplest form of redundancy, called *equivalence induced redundancy*. More specifically, a predicate  $P_i$  is redundant if there exist two propositional formulas in terms of other predicates that are logically equivalent to  $P_i$  and  $\neg P_i$  respectively. The reduced abstract model can be built by replacing  $B_i$  and  $\neg B_i$  using the equivalent formulas. It is easy to see that the resulting reduced abstract model is bisimilar to the original model. We present a method, based on replacement function, to determine if a predicate  $P_i$  can be expressed in terms of the other predicates. The following theorem shows that under some conditions the concretization of the replacement functions for  $B_i$  and  $\neg B_i$  are logically equivalent to  $P_i$  and  $\neg P_i$ .

**Theorem 3.** *For a predicate  $P_i$ , if  $g \mid_{B_i} \wedge g \mid_{\neg B_i} = \emptyset$ , then  $\gamma(\mathcal{F}_U(B_i)) \equiv P_i$  and  $\gamma(\mathcal{F}_U(\neg B_i)) \equiv \neg P_i$ .*

Equivalence induced redundancy occurs often because of a heuristic we use in predicate abstraction. For hardware verification, the predicates we consider are all propositional formulas over concrete state variables with finite domains. It is well known that the number of propositional formulas over  $n$  boolean variables is  $2^{2^n}$ . Therefore, it is possible that the abstract model is much bigger than the concrete model. So we use a heuristic to avoid this problem. Given a predicate, which is a propositional formula, the concrete state variables that this formula depends on are called the *supporting variables* of the predicate. We partition the predicates into clusters. Two predicates go into the same

cluster if they share many supporting variables. Let  $c$  be a cluster. If the number of predicates in  $c$  is greater than the number of supporting variables (non-boolean variables are expanded to bits that are required to encode their domains) in  $c$  then we will use the supporting variables instead of the predicates to build the abstract model. So, all the original predicates in the cluster  $c$  become redundant. In this way, the overall size of the abstract model will be bounded by the size of the concrete model. Since software system may have unbounded state variables, this technique can not be used in general for software verification.

## 6 Redundant Predicates for Safety Properties

A predicate in a given set of predicates is *redundant* for a set of properties in  $\mathcal{L}$  if the abstract transition system constructed without using this predicate satisfies the same set of properties as the original abstract transition system (constructed using all the predicates). In this section we deal with safety properties of the form  $\mathbf{AG} p$ , where  $p$  is a boolean formula without temporal operators. Note that any safety property can be rewritten into the above form through tableau construction with no fairness constraints [9].

Let  $\hat{S}$  be a set of states defined by a set of boolean variables  $V = \{B_1, B_2, \dots, B_k\}$  as before, and  $U = V \setminus \{B_i\}$ . Let  $S_r = \text{proj}[U](\hat{S})$  denote the projection of the set  $\hat{S}$  on  $U$ . For any state  $s_r \in S_r$ ,  $\text{extend}[B_i](s_r)$  is a set of states defined as follows:

- If  $\mathcal{F}_U(B_i)(s_r)$ , then  $\text{extend}[B_i](s_r) = \{\langle s_r, 1 \rangle\}$ .
- If  $\mathcal{F}_U(\neg B_i)(s_r)$ , then  $\text{extend}[B_i](s_r) = \{\langle s_r, 0 \rangle\}$ .
- If  $\neg \mathcal{F}_U(B_i)(s_r) \wedge \neg \mathcal{F}_U(\neg B_i)(s_r)$ ,  $\text{extend}[B_i](s_r) = \{\langle s_r, 0 \rangle, \langle s_r, 1 \rangle\}$ .

We say that a set of consistent abstract states  $\hat{S}$  is *oblivious* to  $B_i$  if and only if

$$\forall \hat{s} \in \hat{S}. (\neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s})) \Rightarrow (\hat{s}[B_i \leftarrow 0] \in \hat{S} \wedge \hat{s}[B_i \leftarrow 1] \in \hat{S})$$

where  $\hat{s}[B_i \leftarrow 0]$  is a state that agrees with  $\hat{s}$  on all bits except possibly the bit  $B_i$ , which is fixed to 0.  $\hat{s}[B_i \leftarrow 1]$  is similar. Intuitively, if neither  $\mathcal{F}_U(B_i)(\hat{s})$  nor  $\mathcal{F}_U(\neg B_i)(\hat{s})$  holds, the values of variables  $B_1, \dots, B_{i-1}, B_{i+1}, \dots, B_k$  can not determine the value of  $B_i$ . In order for  $\hat{S}$  to be oblivious, it must contain states with both possible values of  $B_i$ .

A transition relation  $\hat{R} \subseteq \hat{S} \times \hat{S}$  is called oblivious to  $B_i$ , if for any state  $\hat{s} \in \hat{S}$ ,  $\text{post}[\hat{R}](\hat{s})$  is oblivious to  $B_i$ . More formally,  $\hat{R}$  is oblivious to  $B_i$  if and only if

$$\begin{aligned} \forall \hat{s}, \hat{s}' [ \neg \mathcal{F}_U(B_i)(\hat{s}') \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}') \Rightarrow \\ (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1]) \Leftrightarrow \hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 0])) ] \end{aligned} \quad (8)$$

In order to test whether a transition relation  $\hat{R}$  is oblivious to  $B_i$  or not, we take the negation of (8) and formulate it as a SAT instance by converting it into a CNF formula. If the CNF formula is satisfiable then we conclude that  $\hat{R}$  is not oblivious otherwise it is. The negation of (8) is the following

$$\begin{aligned} \exists \hat{s}, \hat{s}' [ \neg \mathcal{F}_U(B_i)(\hat{s}') \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}') \wedge \\ (\hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 1]) \Leftrightarrow \neg \hat{R}(\hat{s}, \hat{s}'[B_i \leftarrow 0])) ] \end{aligned} \quad (9)$$

**Theorem 4.** *Given an abstract transition system  $\hat{M} = (\hat{S}, \hat{S}_0, \hat{R}, \hat{L})$  which corresponds to a set of predicates  $V$ , and a safety property  $f = \mathbf{AG} p$ , where  $p$  is a propositional formula without temporal operators. Also assume that predicate  $B_i$  is one of the predicates in  $V$  but not one of the predicates in  $f$ . If  $\hat{S}_0$  and  $\hat{R}$  are oblivious to  $B_i$ , then the abstract transition system corresponding to the reduced set of predicates  $U = V \setminus \{B_i\}$  satisfies  $f$  if and only if  $\hat{M}$  satisfies it.*

## 7 Redundant Predicates for Bisimulation Equivalence

In the previous section, the reduced abstract model  $M_r$  was such that it satisfies a safety property, if and only if  $\hat{M}$  satisfies it. We can strengthen this result so that  $M_r$  is bisimulation equivalent to  $\hat{M}$  by imposing slightly different conditions on  $\hat{R}$ .

Let  $\beta \subseteq \hat{S} \times S_r$  be a relation defined such that two states  $\hat{s} \in \hat{S}$  and  $s_r \in S_r$  are related under  $\beta$  if and only if  $\hat{s} \in \text{extend}[B_i](s_r)$ , where  $\text{extend}[B_i](s_r)$  is as defined previously. We intend to make  $\beta$  a bisimulation relation between  $\hat{M}$  and  $M_r$ . From the construction of  $M_r$ , it is easy to see that  $\hat{M} \preceq_\beta M_r$ . In order for  $\hat{M}$  to simulate  $M_r$ , we must make sure that for any  $b_i \in \{0, 1\}$ , if  $\langle s_r, b_i \rangle$  is a consistent abstract state, then  $\langle s_r, b_i \rangle$  can simulate  $s_r$ . If only one of  $\langle s_r, 0 \rangle$  and  $\langle s_r, 1 \rangle$  is a consistent state, from (6), it is easy to see that any successor state of  $s_r$  corresponds to a successor of the single consistent state. In order to handle the case when both  $\langle s_r, 0 \rangle$  and  $\langle s_r, 1 \rangle$  are consistent, we have the following condition on  $\hat{R}$ : for any state  $\hat{s} \in \hat{S}$

$$\neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}) \Rightarrow \forall \hat{s}' . (\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow \hat{R}(\hat{s}[B_i \leftarrow 1], \hat{s}')) \quad (10)$$

This condition says that if the value of  $B_i$  cannot be determined by the values of the other boolean variables, i.e., both  $\hat{s}[B_i \leftarrow 0]$  and  $\hat{s}[B_i \leftarrow 1]$  are consistent, then  $\hat{R}$  does not distinguish between different values of the bit  $B_i$ . If  $\mathcal{F}_U(B_i)(\hat{s})$  is true then we know that  $\hat{s}[B_i \leftarrow 0]$  is inconsistent. If  $\mathcal{F}_U(\neg B_i)(\hat{s})$  is true then we know that  $\hat{s}[B_i \leftarrow 1]$  is inconsistent. In case that both of these are false (which is the condition on the left hand side of (10)), then we require that the successors of the states  $\hat{s}[B_i \leftarrow 0]$ ,  $\hat{s}[B_i \leftarrow 1]$  be the same. Similar to Section 6, to test whether  $\hat{R}$  satisfies condition (10) or not, we test the satisfiability of its negation.

$$\begin{aligned} \exists \hat{s}, \hat{s}' . \quad & \neg \mathcal{F}_U(B_i)(\hat{s}) \wedge \neg \mathcal{F}_U(\neg B_i)(\hat{s}) \wedge \\ & (\hat{R}(\hat{s}[B_i \leftarrow 0], \hat{s}') \Leftrightarrow (\neg \hat{R})(\hat{s}[B_i \leftarrow 1], \hat{s}')) \end{aligned} \quad (11)$$

**Theorem 5.** *If condition (10) holds, then  $\beta$  is a bisimulation relation between  $\hat{M}$  and  $M_r$ .*

It is interesting to note that the conditions for preserving safety properties and bisimulation equivalence are different and do not subsume each other.

## 8 Difference in the Bisimulation and $\mathbf{AG} p$ Conditions

We have seen two redundancy conditions, one for preserving  $\mathbf{AG} p$  properties and the other for preserving  $CTL^*$  properties. In this section, we give examples of transition relation which satisfy one of the conditions and violates the other. This demonstrates that the conditions (8) and (10) are not comparable.

### 8.1 A Transition Relation That Satisfies the Bisimulation Condition

We first present an abstract transition relation that satisfies the Bisimulation condition, (10), but does not satisfy the obliviousness condition required for preserving  $\mathbf{AG} p$  properties. The abstract transition system is:

$$\begin{aligned} (a) \quad & B_2 \rightarrow B'_1 \wedge B'_4 \\ (b) \quad & B_3 \rightarrow B'_1 \wedge B'_2 \\ (c) \quad & B_4 \rightarrow B'_4 \end{aligned}$$

Suppose we are trying to remove  $B_2$ . Assume that  $\mathcal{F}_U(B_2) = \neg B_3$  and  $\mathcal{F}_U(\neg B_2) = \neg B_4$ . The condition for bisimulation, (9), then is

$$B_3 \wedge B_4 \Rightarrow [((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))]$$

If  $B_3 \wedge B_4$  is false then the condition is true. If  $B_3 \wedge B_4$  is true then we need to check the validity of

$$((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)).$$

Now from (b)  $B'_1$  is true if  $B_3$  is true and from (c)  $B'_4$  is true if  $B_4$  is true. So the problem now reduces to validity of

$$((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B_3 \rightarrow B'_1 \wedge B'_2) \wedge (B_4 \rightarrow B'_4))$$

which is trivially true. So  $\hat{R}$  satisfies the bisimulation condition. Now we show that it does not satisfy the condition for  $\mathbf{AG} p$  preservation. Condition for  $\mathbf{AG} p$  preservation in this case would be

$$B'_3 \wedge B'_4 \Rightarrow [((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge 0) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1 \wedge 1) \wedge (B_4 \rightarrow B'_4))]$$

which is equivalent to

$$B'_3 \wedge B'_4 \Rightarrow [((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow false) \wedge (B_4 \rightarrow B'_4)) \Leftrightarrow ((B_2 \rightarrow B'_1 \wedge B'_4) \wedge (B_3 \rightarrow B'_1) \wedge (B_4 \rightarrow B'_4))]$$

This expression is not true for  $B'_3 = B'_4 = B'_1 = B_3 = 1$  and  $B_2 = B_4 = 0$ . So we have shown a transition relation  $\hat{R}$  that satisfies the bisimulation condition but not the  $\mathbf{AG} p$  preservation condition.

## 8.2 A Transition Relation That Satisfies the AG $p$ Condition

The transition relation is

$$\begin{aligned} B_3 &\rightarrow \neg B'_4 \\ B_2 &\rightarrow B'_1 \wedge B'_5 \\ B_3 &\rightarrow B'_1 \wedge \neg B'_2 \end{aligned}$$

We assume that  $\mathcal{F}_U(B_2) = \neg B_3$  and  $\mathcal{F}_U(\neg B_2) = \neg B_4$ . The **AG**  $p$  preservation condition (after some simplification) is

$$B'_3 \wedge B'_4 \Rightarrow [((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (\neg B_3)) \Leftrightarrow ((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1))]$$

If  $B'_3 \wedge B'_4$  is false then the above expression is true. In  $B'_3 \wedge B'_4$  is true, then we can prove the following:

- $((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (\neg B_3)) \Rightarrow ((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1))$ . We only need to prove  $\neg B_3$  implies  $(B_3 \rightarrow B'_1)$ , which is trivially true.
- $((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1)) \Rightarrow ((B_3 \rightarrow \neg B'_4) \wedge (B_2 \rightarrow B'_1 \wedge B'_5) \wedge (\neg B_3))$ . We just need to show that if  $B'_3 \wedge B'_4$  is true and  $(B_3 \rightarrow \neg B'_4)$  is true then  $\neg B_3$ . This is clear since  $B'_4$  is true implies  $\neg B'_4$  is not true. And  $(B_3 \rightarrow false)$  can be true only if  $\neg B_3$  is true.

Hence the **AG**  $p$  preservation condition is satisfied. The bisimulation condition for this example (after some simplification) is:

$$B_3 \wedge B_4 \Rightarrow [((B_3 \rightarrow \neg B'_4) \wedge (B_3 \rightarrow B'_1 \wedge \neg B'_2)) \Leftrightarrow ((B_3 \rightarrow \neg B'_4) \wedge (B'_1 \wedge B'_5) \wedge (B_3 \rightarrow B'_1 \wedge \neg B'_2))]$$

This expression is not true for  $B_3 = B_4 = B'_1 = 1$  and  $B'_5 = B'_4 = B'_2 = 0$ .

Hence we have shown two transition relations such that they satisfy only one of the two preservation conditions. From this we can conclude that the two preservation conditions are not related to each other.

## 9 Removing Redundancy for Software Verification

In this section, we will show how our redundancy removal algorithms can be applied to software (or infinite state systems) verification. Performance of our redundancy removal algorithms depend on efficient computation of the replacement functions  $\mathcal{F}_U(B_i)$  ( $\mathcal{F}_U(\neg B_i)$ ) is important. Recall that, we first calculate the set of consistent abstract states  $g$ , and then use  $g|_{B_i}$  and  $g|_{\neg B_i}$  to define the replacement function. For software verification, predicates may be formulas involving unbounded state variables. Instead of using BDDs to compute  $g$  as in Section 3, we use a theorem prover to compute  $g = \alpha(\mathbf{true})$  as in traditional predicate abstraction [1,21,22]. Computing  $\alpha(\mathbf{true})$  could involve many calls to a theorem prover [22]. However, the correctness of our overall algorithm does not depend on the precise calculation of the set of consistent abstract states. Any approximation,  $f$ , of  $\mathcal{F}_U(B_i)$  satisfying  $\gamma(f) \Rightarrow \gamma(B_i)$  would be sufficient. So the existing techniques for approximating  $\alpha$  can be applied [1].

Except for the replacement function, all the other computations, required in our algorithms to identify and remove redundancy, are performed on the original abstract model. It is usually the case that the abstract model is finite state. Therefore, the algorithms in the previous sections can be easily applied. The *boolean programs* in the SLAM project [1,2], which have infinite control, are an exception. Extending our algorithms to them is left for future work.

## 10 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker and the zChaff SAT solver [23]. The method to compute predicates for all the experiments is based on the deadend and bad states separation algorithm presented in [8].

We present the results for hardware verification in Table 1. We have 6 safety properties to verify for a programmable FIR filter (PFIR) which is a component of a system-on-chip design. For all the properties shown in the first column of Table 1, we have performed cone-of-influence reduction before the verification. The resulting number of registers and gates is shown in the second and third columns. Most properties are true, except for *scr1* and *prop5*. The lengths of the counterexamples are shown in the fourth column. All these properties are difficult to verify for the state-of-art BDD-based model checker, Cadence SMV. Except for the two false properties, Cadence SMV can not verify any of them in 24 hours. The verification time for *scr1* is 834 seconds, and for *prop5* is 8418 seconds, which are worse than our results.

We compare the systems we built with and without the redundancy removal techniques described in this paper. In Table 1, the fifth to seventh columns are the results obtained without our techniques; while the last four columns are the results obtained with the techniques enabled. We compare the time (in seconds), the number of refinement iterations, and the number of predicates in the final abstraction. The last column is the number of redundant predicates our method is able to identify. In all cases, our new method outperforms the old one in the amount of time used, sometimes over an order of magnitude improvement is achieved. With the new method, the number of refinement iterations is usually smaller. We can usually identify a significant number of predicates as redundant. As a result, the number of predicates in the final abstraction is usually small.

**Table 1.** Comparison without and with redundancy removal

circuit	# regs	# gates	ctrex length	Old			New			
				time	iters	pred	time	iters	pred	red
<i>scr1</i>	243	2295	16	637.5	103	40	386.4	67	34	23
<i>prop5</i>	250	2342	17	2262.0	131	48	756.2	101	44	26
<i>prop8</i>	244	2304	true	288.5	68	35	159.8	40	25	20
<i>prop9</i>	244	2304	true	2448.7	146	46	202.7	43	27	25
<i>prop10</i>	244	2304	true	6229.3	161	55	178.2	50	25	23
<i>prop12</i>	247	2317	true	707.0	111	45	591.2	80	38	26

## 11 Conclusion and Future Work

We have presented new algorithms for the identification and removal of redundant predicates. These algorithms enable us to identify three conditions for redundancy removal: equivalence induced redundancy, redundancy that preserves safety properties and redundancy that preserves bisimulation equivalence. Once a redundant predicate has been identified, a reduced abstract model can be efficiently computed without referring to the concrete model. Experimental results demonstrate the usefulness of our algorithms.

An interesting extension of the work presented in this paper is to identify redundant *sets of predicates*. That is, instead of identifying redundant predicates one at a time, sets of redundant predicates are identified together. In this setting the redundancy criteria may have to be different. The presented algorithms work for both hardware and software verification provided that the abstract model is finite state. The SLAM project uses *boolean programs* as the abstract model, which might have infinite control. It will be interesting to investigate how to extend our algorithm to handle boolean programs.

## References

1. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *PLDI*, 2001.
2. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstractions for model checking c programs. In *TACAS 2001*, volume 2031 of *LNCS*, pages 268–283, April 2001.
3. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427, pages 319–331, Vancouver, Canada, 1998. Springer-Verlag.
4. Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
5. Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Computer Science and Applied Mathematics Series. Academic Press, New York, NY, 1973.
6. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *Proc. of the 19th Annual Symposium on Principles of Programming Languages*, pages 343–354, 1992.
7. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided Abstraction Refinement. In *Twelfth Conference on Computer Aided Verification (CAV'00)*. Springer-Verlag, July 2000.
8. Edmund Clarke, Muralidhar Talupur, and Dong Wang. SAT based Predicate Abstraction for Hardware Verification. In *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.
9. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
10. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs, volume 131 of Lect. Notes in Comp. Sci.*, pages 52–71, 1981.
11. Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.
12. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

13. Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *TACAS'03*, 2003.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages 58–70, 2002.
15. Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference*, pages 266–271, 1993.
16. Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, pages 98–112, 2001.
17. 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: An International Journal*, 6(1):11–44, January 1995.
18. K.L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *TACAS'03*, 2003.
19. Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *12th Conference on Computer Aided Verification*, number 1855 in LNCS, 2000.
20. Corina Pasareanu, Matthew Dwyer, and Willem Visser. Finding feasible counter-examples when model checking abstracted java programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, 2001.
21. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
22. H. Saidi and N. Shankar. Abstract and model check while you prove. In *11th Conference on Computer-Aided Verification*, volume 1633 of LNCS, pages 443–454, July 1999.
23. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, November 2001.



# A Symbolic Approach to Predicate Abstraction\*

Shuvendu K. Lahiri<sup>1</sup>, Randal E. Bryant<sup>1</sup>, and Byron Cook<sup>2</sup>

<sup>1</sup> Carnegie Mellon University, Pittsburgh, PA  
shuvendu@ece.cmu.edu, Randy.Bryant@cs.cmu.edu

<sup>2</sup> Microsoft Corporation, Redmond, WA  
bycook@microsoft.com

**Abstract.** Predicate abstraction is a useful form of abstraction for the verification of transition systems with large or infinite state spaces. One of the main bottlenecks of this approach is the extremely large number of decision procedure calls that are required to construct the abstract state space. In this paper we propose the use of a symbolic decision procedure and its application for predicate abstraction. The advantage of the approach is that it reduces the number of calls to the decision procedure exponentially and also provides for reducing the re-computations inherent in the current approaches. We provide two implementations of the symbolic decision procedure: one based on BDDs which leverages the current advances in early quantification algorithms, and the other based on SAT-solvers. We also demonstrate our approach with quantified predicates for verifying parameterized systems. We illustrate the effectiveness of this approach on benchmarks from the verification of microprocessors, communication protocols, parameterized systems, and Microsoft Windows device drivers.

## 1 Introduction

Abstraction is crucial in the verification of systems that have large data values, memories, and parameterized processes. These systems include microprocessors with large data values and memories, parameterized cache coherence protocols, and software programs with arbitrarily large values. *Predicate abstraction*, first proposed by Graf and Saidi [15] as a special case of the general framework of *abstract interpretation* [10], has been used in the verification of protocols [15,22], parameterized systems [11,12] and software programs [1,13]. In predicate abstraction, a finite set of predicates is defined over the concrete set of states. These predicates are used to construct a finite state abstraction of the concrete system. The automation in generating the finite abstract model makes this scheme attractive in combining deductive and algorithmic approaches for infinite state verification.

One of the main problems in predicate abstraction is that it typically makes a large number of theorem prover calls when computing the abstract transition relation or the abstract state space. Most of the current methods, including

---

\* This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.001.

Saidi and Shankar [22], Das, Dill and Park [12], Ball et al. [1], Flanagan and Qadeer [13] require a number of validity checks that can be exponential in the number of predicates.

A number of tools [1,15] address this problem by only approximating the abstract state space — resulting in a weaker abstract transition relation. Das and Dill [11] have proposed *refining* the abstract transition relation based on counterexamples, starting with an initial transition relation. This technique can work well — particularly for systems with sparse abstract state graphs. However, this technique may still require a potentially exponential number of calls to a decision procedure. Namjoshi and Kurshan [20] have proposed an alternative technique by syntactically transforming a concrete system to an abstract system. Instead of using a theorem prover, they propose the use of syntactic strategies to eliminate first-order quantifiers. However, the paper does not report empirical results to demonstrate the effectiveness of the approach.

In this work, we present a technique to perform predicate abstraction that reduces the number of calls to decision procedures exponentially. We formulate a symbolic representation of the predicate abstraction step, reduce it to a quantified Boolean formula and then use Boolean techniques (based on Binary Decision Diagrams (BDDs) [4] and Boolean Satisfiability solvers (SAT)) to generate a symbolic representation of the abstract transition relation or abstract state space. Our work is motivated by the recent work in UCLID [6,24], which transforms quantifier-free first-order formulas into Boolean formulas and uses Boolean techniques to solve the resulting problems.

The advantage of our approach is three-fold: First, we can leverage efficient Boolean quantification algorithms [14] based on BDDs and recent advances in SAT-based techniques for quantification [9,18]. Second, the single call to the symbolic decision procedure eliminates the overhead of multiple (potentially exponential) calls to decision procedures (e.g., initializing data structures, libraries, system calls in certain cases). Third, in previous work, the decision procedures cannot exploit the *pruning* across different calls and result in a lot of re-computation. The *learning* and *pruning* mechanisms in modern SAT solvers allow us to prevent the re-computations.

Our work considers both quantifier-free and quantified predicates. The quantified predicates are used in the verification of parameterized systems and systems with unbounded resources. Experimental results indicate that our method outperforms previous methods by orders of magnitude on a large set of examples drawn from the verification of microprocessors, communication protocols, parameterized systems and Microsoft Windows device drivers.

The paper is organized as follows. In Section 2, we provide some background on predicate abstraction. Section 3 describes how predicate abstraction can be implemented using our symbolic decision procedure. We also describe several ways of implementing the decision procedure. Section 4 describes the handling of universally quantified predicates for verifying parameterized systems. Section 5 presents the results of our experiments.

## 2 Background

Fig. 1 displays the syntax of the Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), a fragment of first-order logic extended with equality, inequality, and counters. An *expression* in CLU can evaluate to truth values (*bool-expr*), integers (*int-expr*), functions (*function-expr*) or predicates (*predicate-expr*). The logic can be used to describe systems in the tool UCLID [6].

$$\begin{aligned}
 \text{bool-expr} &::= \mathbf{true} \mid \mathbf{false} \mid \text{bool-symbol} \\
 &\quad \mid \neg \text{bool-expr} \mid (\text{bool-expr} \wedge \text{bool-expr}) \\
 &\quad \mid (\text{int-expr} = \text{int-expr}) \mid (\text{int-expr} < \text{int-expr}) \\
 &\quad \mid \text{predicate-expr}(\text{int-expr}, \dots, \text{int-expr}) \\
 \text{int-expr} &::= \text{int-var} \mid \text{int-symbol} \\
 &\quad \mid \text{ITE}(\text{bool-expr}, \text{int-expr}, \text{int-expr}) \\
 &\quad \mid \text{int-expr} + \text{int-constant} \\
 &\quad \mid \text{function-expr}(\text{int-expr}, \dots, \text{int-expr}) \\
 \text{predicate-expr} &::= \text{predicate-symbol} \mid \lambda \text{int-var}, \dots, \text{int-var} . \text{bool-expr} \\
 \text{function-expr} &::= \text{function-symbol} \mid \lambda \text{int-var}, \dots, \text{int-var} . \text{int-expr}
 \end{aligned}$$

**Fig. 1. CLU Expression Syntax.** Expressions can denote computations of Boolean values, integers, or functions yielding Boolean values or integers.

A system description is a four-tuple<sup>1</sup>  $(\mathcal{S}, \mathcal{K}, \delta, q^0)$  where:

- $\mathcal{S}$  is a set of symbols denoting the state elements.
- $\mathcal{K}$  is a set of symbols denoting the parameters of the system.
- $\delta$  represents the transition function for each state element in  $\mathcal{S}$ , as CLU expressions over  $\mathcal{S} \cup \mathcal{K}$ .
- $q^0$  represents the set of initial state expressions for each state element in  $\mathcal{S}$ , as CLU expressions over  $\mathcal{K}$ .

State variables can be integers, Booleans, functions over integers or predicates over integers. The functions and predicates represent unbounded mutable arrays of integers and Booleans, respectively. The symbols in  $\mathcal{K}$  are the parameters for the system, and can be used to denote the size of a buffer in the system, the functionality of an ALU in a processor, or the number of processes in a protocol. They can also specify a set of initial states of the system. The logic has been used to model and verify out-of-order processors with unbounded resources and parameterized cache-coherence protocols [6,17].

For a set of symbols  $\mathcal{U}$ , an interpretation  $\sigma_{\mathcal{U}}$  assigns type-consistent values to each of the symbols in the set  $\mathcal{U}$ . For any expression  $\Psi$  over  $\mathcal{U}$ ,  $\langle \Psi \rangle_{\sigma_{\mathcal{U}}}$  denotes the evaluation of the expression under  $\sigma_{\mathcal{U}}$ .

<sup>1</sup> We can encode inputs to the system at each step by a *generator of arbitrary values* as described in previous work by Velev and Bryant [25].

If  $q$  denotes a set of expressions (one for each state element), then  $\langle q \rangle_{\sigma_K}$  applies  $\sigma_K$  point-wise to each expression of  $q$ . If  $\Psi$  represents an expression, then  $\Psi[Y/\mathcal{U}]$  substitutes the expressions in  $Y$  point-wise for the symbols in  $\mathcal{U}$ .

A state  $s$  of the concrete system is an interpretation to the symbols in  $\mathcal{S}$ . Given an interpretation  $\sigma_K$  to the symbols in  $\mathcal{K}$ , the initial state of the concrete system is given as  $\langle q^0 \rangle_{\sigma_K}$ , and an *execution sequence* of the concrete system is given by  $\langle q^0 \rangle_{\sigma_K}, \dots, \langle q^i \rangle_{\sigma_K}, \langle q^{i+1} \rangle_{\sigma_K}, \dots$ , where  $q^{i+1} = \delta[q^i/\mathcal{S}]$ . A concrete state  $s$  is *reachable* if it appears in an execution sequence for any interpretation  $\sigma_K$ .

For the rest of the paper, we will represent a set of states as either a set or by a Boolean expression over  $\mathcal{S} \cup \mathcal{K}$ .

## 2.1 Predicate Abstraction

The predicate abstraction algorithm generates a finite state abstraction from a large or infinite state system. The finite model can then be used in the place of the original system when performing model checking or deriving invariants. Let  $\Phi \doteq \{\phi_1, \dots, \phi_k\}$  be the set of predicates which are used for inducing the abstract state space. Each of these predicates is a Boolean expression over the set of symbols in  $\mathcal{S} \cup \mathcal{K}$ . Let  $\mathcal{B} \doteq \{b_1, \dots, b_k\}$  denote the Boolean variables in the abstract system such that value of  $b_i$  denotes the evaluation of the predicate  $\phi_i$  over the concrete state. An abstract state  $s_a$  is an interpretation to the variables in  $\mathcal{B}$ . The *abstraction* and the *concretization* functions  $\alpha$  and  $\gamma$  are defined over sets of concrete and abstract states respectively.

If  $\Psi_c$  describes a set of concrete states (as an expression over  $\mathcal{S} \cup \mathcal{K}$ ), then:

$$\alpha(\Psi_c) \doteq \{s_a \mid \exists s_c \in \Psi_c \exists \sigma_K \text{ s.t. } \bigwedge_{i \in 1, \dots, k} \langle b_i \rangle_{s_a} \Leftrightarrow \langle \phi_i \rangle_{s_c \cup \sigma_K}\}$$

If  $\Psi_a$  represents a set of abstract states, then

$$\gamma(\Psi_a) \doteq \{s_c \mid \alpha(\{s_c\}) \in \Psi_a\}$$

In fact, the concretization function  $\gamma$  is implemented as the substitution,  $\gamma(\Psi_a) \doteq \Psi_a[\Phi/\mathcal{B}]$ . For each predicate  $\phi_i$ , let  $\phi'_i$  represent the result of substituting the next-state expression for each symbol in  $\mathcal{S}$ . That is,  $\phi'_i \doteq \phi_i[\delta(\mathcal{S})/\mathcal{S}]$ . Let  $\Phi' \doteq \{\phi'_1, \dots, \phi'_k\}$ .

Let us now define a *cube* over  $\mathcal{B}$ ,  $c_B$ , to be a clause  $l_{i_1} \wedge \dots \wedge l_{i_m}$ , where each  $l_{i_j}$  is a *literal* (either  $b_{i_j}$  or  $\neg b_{i_j}$ , where  $b_{i_j} \in \mathcal{B}$ ). A *cube* over  $\mathcal{B}$  is *complete* if all the symbols in  $\mathcal{B}$  are present as literals in the cube. A complete cube over  $\mathcal{B}$  corresponds to an abstract state.

Now, we define the computation of the abstract state space using the current methods [12,22]. To obtain the set of abstract initial states,  $\Psi_B^0$ , the cubes over  $\mathcal{B}$  are enumerated and a complete cube  $c_B$  is included in  $\Psi_B^0$  iff the expression  $\gamma(c_B)[q^0/\mathcal{S}]$  is satisfiable. The set of reachable abstract states are computed by performing a fixpoint computation, starting from  $\Psi_B^0$  and computing  $\Psi_B^1, \dots, \Psi_B^r$ , the states reachable within  $1, \dots, r$  steps from  $\Psi_B^0$  until  $\Psi_B^{r+1} \Rightarrow \Psi_B^r$ . The main operation in this process is the computation of the set of abstract successor states  $\Psi'_B$ , for a set of states  $\Psi_B$ .

Current methods compute  $\Psi'_B$  iteratively by enumerating *cubes over*  $\mathcal{B}$  and including a complete cube  $c_B$  in  $\Psi'_B$ , iff the expression  $\gamma(\Psi_B) \wedge c_B [\Phi'/\mathcal{B}]$  is satisfiable. The satisfiability is checked using decision procedures for (quantifier-free) first-order logic. Since an exponential number of cubes over  $\mathcal{B}$  have to be enumerated in the worst case, a very large number of decision procedure calls are involved in the computation of the abstract state space.

In the next section, we demonstrate how to avoid the possibly exponentially number of calls to decision procedure by providing a symbolic method to obtain the set of initial states  $\Psi_B^0$  and the set of successor states  $\Psi'_B$ , given  $\Psi_B$ .

### 3 Symbolic Predicate Abstraction

Decision procedures for CLU [6,24] translate a CLU formula  $F_{clu}$  to a propositional formula  $\widetilde{F}_{clu}$ , such that  $F_{clu}$  is satisfiable iff  $\widetilde{F}_{clu}$  is satisfiable. Different methods have been proposed for translating a CLU formula to the propositional formula. These methods differ in ways to enforce the constraints for various theories (uninterpreted functions, inequality, equality etc.) to construct the final propositional formula. We will give intuitive description of the different methods as required in the different parts of the paper. Details of the procedures are outside the scope of this paper, and interested readers are referred to previous work on this subject [5,6,24].

Now, for every Boolean subexpression  $E$  of  $F_{clu}$ , let  $\widetilde{E}$  denote the *corresponding* Boolean subexpression in the final propositional formula (if it exists). The final formula is denoted as  $\widetilde{F}_{clu}$ <sup>2</sup>. Let  $\Sigma$  be the set of symbols in the CLU formula  $F_{clu}$  (can include Boolean, integer and function symbols) and  $\widetilde{\Sigma}$  be the set of (Boolean) symbols in the final formula  $\widetilde{F}_{clu}$ , where the set of Boolean symbols in  $\widetilde{\Sigma}$  include all the Boolean symbols in  $\Sigma$ . The different Boolean encoding methods preserve the valuation of the Boolean symbols in the original formula:

**Proposition 1.** *There exists an interpretation  $\sigma$  over  $\Sigma$ , such that  $\langle F_{clu} \rangle_\sigma$  is **true**, iff there exists an interpretation  $\sigma'$  over  $\widetilde{\Sigma}$ , such that  $\langle \widetilde{F}_{clu} \rangle_{\sigma'}$  is **true** and for every Boolean symbol  $b$  in  $F_{clu}$ ,  $\langle b \rangle_\sigma \Leftrightarrow \langle b \rangle_{\sigma'}$ .*

Now consider a CLU formula  $F$  over the symbols  $\Sigma \cup \mathcal{A}$ , where  $\mathcal{A}$  contains Boolean symbols only and  $\Sigma \cap \mathcal{A} = \{\}$ . The Boolean formula  $\widetilde{F}$  is a formula over  $\widetilde{\Sigma} \cup \mathcal{A}$ , where  $\widetilde{\Sigma}$  are the Boolean symbols in the final propositional formula other than symbols in  $\mathcal{A}$ . Using proposition 1, we can show the following:

**Proposition 2.** *For any interpretation  $\sigma_{\mathcal{A}}$  over the symbols in  $\mathcal{A}$ ,  $\langle \exists \Sigma : F \rangle_{\sigma_{\mathcal{A}}} \Leftrightarrow \langle \exists \widetilde{\Sigma} : \widetilde{F} \rangle_{\sigma_{\mathcal{A}}}$ .*

Now let us get back to the problem of obtaining the initial set of abstract states  $\Psi_B^0$  and obtaining the expression for the successors,  $\Psi'_B$  for the set of abstract

<sup>2</sup> There can be a Boolean subexpression  $E_1$  in  $F_{clu}$  which may not have a corresponding Boolean subexpression  $\widetilde{E}_1$  in  $\widetilde{F}_{clu}$ . This can arise because of optimizations like  $(\mathbf{true} \vee E_1 \longrightarrow \mathbf{true})$ .

states in  $\Psi_{\mathcal{B}}$ . The set of initial states for the abstract system is given by the expression

$$\Psi_{\mathcal{B}}^0 \doteq \exists \mathcal{K} : \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \phi_i [q^0 / \mathcal{S}] \quad (1)$$

Similarly, given  $\Psi_{\mathcal{B}}$ , one can obtain the set of successors using the following equation:

$$\Psi'_{\mathcal{B}} \doteq \exists \mathcal{S} : \exists \mathcal{K} : \gamma(\Psi_{\mathcal{B}}) \wedge \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \phi'_i \quad (2)$$

The correctness of the encodings follows from the following proposition:

**Proposition 3.** *For any complete cube  $c_{\mathcal{B}}$  over  $\mathcal{B}$ ,  $c_{\mathcal{B}} \Rightarrow \Psi_{\mathcal{B}}^0$  iff the expression  $\gamma(c_{\mathcal{B}}) [q^0 / \mathcal{S}]$  is satisfiable. Similarly, for any complete cube  $c_{\mathcal{B}}$  over  $\mathcal{B}$ ,  $c_{\mathcal{B}} \Rightarrow \Psi'_{\mathcal{B}}$  iff the expression  $\gamma(\Psi_{\mathcal{B}}) \wedge c_{\mathcal{B}} [\Phi' / \mathcal{B}]$  is satisfiable.*

Notice that in Equations 1 and 2, the only free variables are the set of Boolean symbols  $\mathcal{B}$ . Hence one can use Proposition 2 to reduce these second-order formulas (there can be function symbols in  $\mathcal{S} \cup \mathcal{K}$ ) to an existentially quantified Boolean formula. For example, the result of propositional encoding of the formula in Equation 2 (with  $\Sigma \doteq \mathcal{S} \cup \mathcal{K}$ ) yields the following structure<sup>3</sup>:

$$\exists \tilde{\Sigma} : C \wedge \gamma(\widetilde{\Psi_{\mathcal{B}}}) \wedge \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \tilde{\phi}'_i \quad (3)$$

where  $C$  denotes a set of propositional constraints over the symbols in  $\tilde{\Sigma}$  to enforce the semantics of different first-order theories (uninterpreted functions, equality, inequality etc.).

This formulation is also applicable to predicate abstraction techniques which derive the *Weakest Boolean Precondition* (*WBP*) [1] over the set of predicates  $\Phi$ , given the *weakest precondition*  $WP(\delta, \Psi_{\mathcal{S}})$  of a CLU expression  $\Psi_{\mathcal{S}}$  with respect to the transition function  $\delta$ . In this case, a cube  $c_{\mathcal{B}}$  is included in *WBP* if  $c_{\mathcal{B}} [\Phi / \mathcal{B}] \Rightarrow WP(\delta, \Psi_{\mathcal{S}})$  is *valid*. The set of cubes which are not included in *WBP* is given as:

$$\overline{WBP} \doteq \exists \mathcal{S} : \exists \mathcal{K} : \neg WP(\delta, \Psi_{\mathcal{S}}) \wedge \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \phi_i \quad (4)$$

and  $WBP \doteq \neg \overline{WBP}$ .

Similarly, one can obtain a symbolic expression for the abstract transition relation as:

$$\delta(\mathcal{B}, \mathcal{B}') \doteq \exists \mathcal{S} : \exists \mathcal{K} : \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \phi_i \wedge \bigwedge_{i \in \{1, \dots, k\}} b'_i \Leftrightarrow \phi'_i \quad (5)$$

<sup>3</sup> The translation needs to ensure that  $(\neg \phi'_i)$  is syntactically same as  $\neg(\phi'_i)$ . This requirement is violated for certain optimizations that push  $\neg$  to the leaves of the formula [23]. Hence we have to disable such transformations in the Boolean translation.

*Example 1.* Consider a concrete system with two integer state variables,  $x$  and  $y$ . Thus  $\mathcal{S} = \{x, y\}$ . The initial state  $q^0$  is given as  $q_x^0 = c_0$  and  $q_y^0 = c_0$ . The transition function  $\delta$  is given as  $\delta_x \doteq f(y)$  and  $\delta_y \doteq f(x)$ . The only parameters which appear in this system are  $\mathcal{K} = \{c_0, f\}$ . Consider the predicate  $\phi_1 \doteq x = y$  over the system.

The initial abstract state is given as  $\Psi_B^0 \doteq \exists c_0 : b_1 \Leftrightarrow (c_0 = c_0) \doteq b_1$ .

For the first iteration,  $\Psi_B \doteq \Psi_B^0$ . Now  $\gamma(b_1) \doteq x = y$  and  $\phi'_1 \doteq (f(y) = f(x))$ . The set of successor states  $\Psi'_B$  is denoted as

$$\exists x : \exists y : \exists f : x = y \wedge b_1 \Leftrightarrow (f(y) = f(x))$$

We eliminate the function symbols in the formula by the method of Bryant et al. [5].  $f(y)$  is replaced by a fresh symbolic constant  $vf_1$ .  $f(x)$  is replaced by the expression  $ITE(x = y, vf_1, vf_2)$  to preserve functional consistency. After eliminating the  $ITE$ , the equation becomes

$$\exists x : \exists y : \exists vf_1 : \exists vf_2 : x = y \wedge b_1 \Leftrightarrow (x = y \wedge vf_1 = vf_1 \vee \neg(x = y) \wedge vf_1 = vf_2)$$

For this example, we use the encoding of inequalities using separation predicates (SEP) method [24]. Each inequality is encoded with a fresh Boolean variable and transitivity constraints are imposed. We use the Boolean variables  $e_{x,y}$  to encode  $x = y$  and Boolean variable  $e_f$  to encode  $vf_1 = vf_2$ . In this case, no transitivity constraints are required. The quantified Boolean formula for  $\Psi'_B$  becomes

$$\Psi'_B \doteq \exists e_{x,y}, e_f : e_{x,y} \wedge b_1 \Leftrightarrow (e_{x,y} \vee \neg e_{x,y} \wedge e_f)$$

The solution to this equation is simply  $b_1$ . The set of successor states after the first iteration,  $\Psi_B^1 \doteq \Psi'_B$ . The reachability analysis terminates since  $\Psi_B^0 \Leftrightarrow \Psi_B^1$  and the set of reachable abstract state is given as  $b_1$ .

We have thus reduced the problem of computing the set of initial abstract states  $\Psi_B^0$  and the set of successors  $\Psi'_B$  to the problem of computing the set of solutions to an existentially quantified Boolean formula. In the next few sections, we shall exploit the structure of the formula to efficiently compute the set of solutions. For the rest of the discussion, we assume that we want to solve the existentially quantified Boolean formula  $\exists \tilde{\Sigma} : \tilde{F}$ .

### 3.1 Using BDD-Based Approaches

We can obtain the set of solutions to the quantified Boolean formula  $\exists \tilde{\Sigma} : \tilde{F}$  by constructing the BDD for  $\tilde{F}$  and then existentially quantifying out the variables in  $\tilde{\Sigma}$  using BDD quantification operators. The resultant BDD is a symbolic representation of the formula  $\exists \tilde{\Sigma} : \tilde{F}$ .

The naive method of constructing the BDD for  $\tilde{F}$  and then quantifying out the symbols in  $\tilde{\Sigma}$  is very inefficient, since the size of the intermediate BDD representing  $\tilde{F}$  could be very large. We show that we can exploit the syntactic formula structure to leverage most the efficient quantification techniques (e.g. early quantification [14]) from image computation in symbolic model checking.

Equation 3 resembles the equation for post-image computation in symbolic model checking where  $C \wedge \gamma(\widetilde{\Psi_B})$  represents the set of current states and the tran-

sition relation for the state variable  $b_i$  is given as  $\tilde{\phi}'_i$ . We can treat all the symbols in  $\tilde{\Sigma}$  as present state variables. In fact, the next state for the variables in  $\tilde{\Sigma}$  are left unconstrained and can also be interpreted as inputs. We use NuSMV's [8] INIT expression to specify  $C \wedge \gamma(\tilde{\Psi}_{\mathcal{B}})$  as the set of initial states and for each state variable  $b_i$ , specify the next-state transition as  $\tilde{\phi}'_i$ . We then perform *one* step of post image computation to obtain the set of successor states.

### 3.2 Using SAT-Based Approaches

The main difference between the image computation step in traditional model checking and the quantified equation in Equation 3 is that the number of bound variables (in  $\tilde{\Sigma}$ ) often exceeds the number of variables in  $\mathcal{B}$ . These variables arise because of the boolean encoding of integers in the CLU formula. In many cases, the number of variables in  $\mathcal{B}$  is only 5% of the variables in  $\tilde{\Sigma}$  (for instance 400 Boolean symbols in  $\tilde{\Sigma}$  for just 3 predicates). In these cases, the cost of constructing the BDD and quantifying the variables in  $\tilde{\Sigma}$  is expensive. Instead we can use SAT-based methods to compute the set of solutions to Equation 3.

SAT-based quantification engines [18,9] enumerate solutions over  $\tilde{\Sigma} \cup \mathcal{B}$  for the expression  $\tilde{F}$ . Since we are only interested in the interpretations of symbols in  $\mathcal{B}$ , the part of the assignment which corresponds to symbols in  $\tilde{\Sigma}$  is projected out. To prevent the SAT checker from computing the same assignment to the symbols in  $\mathcal{B}$  again, a *blocking clause* over the variables in  $\mathcal{B}$  is added to the set of conflict clauses to block this assignment. The most challenging part in the entire procedure is to find a minimal blocking clause, which assigns values to a minimal subset of literals over  $\mathcal{B}$ . We have integrated one such tool, SATMC developed by Daniel Kroening [9], as the SAT-based quantification engine. It uses heuristics to efficiently add blocking clauses for the variables in  $\mathcal{B}$  using the data structures and algorithms of ZChaff [19]. We omit the details of the procedure from this paper.

Although SAT-based quantification uses an enumeration technique, there are several advantages over current approaches which use a decision procedure repeatedly to obtain the set of cubes. First, we can take advantage of the *learning* from the SAT solvers, since the same data structure is maintained throughout the computation. Secondly, the ability to perform non-chronological backtracking provides more flexibility to obtain better cubes than the approach in Das, Dill and Park [12], where the order of variables involved in splitting the cubes is fixed. Lastly, we can remove the overhead of invoking the decision procedure repeatedly to obtain the set of solutions.

## 4 Universally Quantified Predicates

To verify systems with function or predicate state elements, we need the ability to specify quantified predicates. The function and predicate state elements allow us to model unbounded arrays of integers, truth values or enumerated types. These unbounded arrays can be used to model memories, queues or network of



identical processes. For example, if  $pc$  is a state element which maps an integer to an enumerated set of states, then  $pc(i)$  denotes the state of the  $i^{th}$  process in the system. Properties for such parameterized systems are expressed as quantified formulas. To state the property of mutual exclusion, one has to state  $\forall i, j : i \neq j \Rightarrow \neg(pc(i) = \text{CS} \wedge pc(j) = \text{CS})$ , where  $\text{CS}$  is the state of a process in the critical section.

However, introducing universally quantified predicates (predicates of the form  $\forall \vec{i} : \phi(\vec{i})$ , where  $\phi(\vec{i})$  is a quantifier-free CLU expression and  $\vec{i}$  is a vector of integer variables) adds two dimensions to the problem:

1. For quantifier-free predicates, concretization of the reachable abstract state space yields the strongest expression involving the predicates (Boolean combination of the predicates) that approximates the concrete reachable states. This expression serves as an invariant for the system. For parameterized systems, we have found that the inductive invariant can often be expressed as  $\forall \vec{i} : P(\vec{i})$ , where  $P$  is a quantifier-free CLU expression [17,21]. However, given quantified predicates  $\forall \vec{i} : \phi_1(\vec{i}), \dots, \forall \vec{i} : \phi_k(\vec{i})$ , a Boolean combination of the predicates does not always yield the strongest expression of the form  $\forall \vec{i} : P(\vec{i})$ , where  $P(\vec{i})$  is a Boolean combination of  $\phi_1(\vec{i}), \dots, \phi_k(\vec{i})$ . For example, one cannot obtain the expression  $\forall i : \phi_1(i) \vee \phi_2(i)$  using a combination of the predicates  $\forall i : \phi_1(i)$  and  $\forall i : \phi_2(i)$ .
2. Introducing universally quantified predicates requires us to check satisfiability of first-order formulas with both universal and existential quantifiers, which is an undecidable task. Hence we need sound quantifier instantiation strategies to eliminate the universal quantifiers.

To address the first problem, the user provides the set of predicates  $\Phi \doteq \{\phi_1(\vec{i}), \dots, \phi_k(\vec{i})\}$  with an implicit quantifier over  $\vec{i}$  (similar to the work by Flanagan and Qadeer [13]). As before, we associate a Boolean variable  $b_i$  with  $\phi_i(\vec{i})$ . If  $\Psi_{\mathcal{B}}$  be an expression over the symbols in  $\mathcal{B}$ , then Equation 2 can be written as:

$$\Psi'_{\mathcal{B}} \doteq \exists \mathcal{S} : \exists \mathcal{K} : (\forall \vec{i} : \Psi_{\mathcal{B}} [\Phi(\vec{i})/\mathcal{B}]) \wedge \exists \vec{j} : \bigwedge_{i \in \{1, \dots, k\}} b_i \Leftrightarrow \phi'_i(\vec{j}) \quad (6)$$

Now we need to address the second problem. First, the existential quantifiers over  $\vec{j}$  are pulled outside. The universal quantifiers over  $\vec{i}$  are instantiated using sound instantiation techniques present in UCLID [17]. The resulting formula has the same existentially quantified structure as that of Equation 2 and thus can be solved using BDD or SAT-based quantification as before.

Finally, if  $\Psi_{\mathcal{B}}^*$  is the set of reachable states over  $\mathcal{B}$ , then the invariant of the concrete system is  $\forall \vec{i} : \Psi_{\mathcal{B}}^* [\Phi(\vec{i})/\mathcal{B}]$ .

## 5 Results

We have built a prototype of the methods discussed into the tool UCLID [6]. To compare the effectiveness of the approach, we compare against an implementation of a recursive case-splitting based approach suggested by Das, Dill and

Park [12]. The approach is based on checking satisfiability of individual cubes as mentioned in Section 2. The Stanford Validity Checker<sup>4</sup> (SVC) [2] is used as the decision procedure for checking first order formulas. Various optimizations (such as considering cubes in the increasing order of lengths) are performed to reduce the number of calls to the decision procedure from the exponential worst case.

We have analyzed the different ways to encode the first order formula as a Boolean formula. In the rest of the discussion, we use the function elimination strategy by Bryant et al. [5]. Integers are encoded using two methods:

- FI : The domain of each integer is restricted to a finite but large enough set of values that preserve satisfiability [6].
- SEP : Each separation predicate  $x \bowtie y + c$  is encoded as a Boolean variable and transitivity constraints are imposed [24].

NuSMV is used as the BDD-based model checker. We use dynamic variable ordering with the *sifting* heuristics and IWLS95 heuristics for quantifier scheduling. SATMC is the SAT-based model checker and is used to solve the quantification step. All the experiments were performed on a 2.2 GHz Pentium 4 machine with 900MB of memory.

**Hardware Benchmarks** We have used the predicate abstraction engine in the context of verification of pipelined DLX processor, an unbounded out-of-order processor (OOO), communication protocols and mutual exclusion protocols. Below we describe some of them.

Predicate abstraction was used to approximate the reachable set of states for the DLX pipelined processor. This is used to restrict the set of initial states for the Burch and Dill commutative diagram approach [7]. Current processor verification methods that employ Burch and Dill’s technique [5,16] requires the user to manually provide invariants to restrict the most general state of the system. The approximate state space was also used to verify the absence of data-hazards and correctness of stalling logic for the processor.

Quantified predicates are used to derive candidate invariants for the deductive verification of an out-of-order processor with unbounded resources [17]. Various infinite-state protocols have also been chosen to demonstrate the effectiveness of the approach. We have chosen the two process Bakery algorithm (Bakery-2) and a parameterized semaphore protocol [21]. We also chose a communication protocol called the Bounded Retransmission Protocol (BRP) which was described by Graf and Saidi [15] for demonstrating the use of predicate abstraction.

Fig 2 illustrates the performance of the different approaches on a set of benchmarks. For DLXb, the number of calls to SVC reported are even before the first iteration for the explicit version completes.

---

<sup>4</sup> We have experimented with other decision procedures (e.g. CVC [3], UCLID [6]) but have not found any significant improvement for this application.

Example	Preds	Iter	Explicit Method		Symbolic Time (sec)			
			SVC Time	# of Calls	SATMC-based		NuSMV-based	
					FI	SEP	FI	SEP
Semaphore	8	5	37.0	513	16.3	9.7	5.1	2.6
DLXa	5	3	24.5	199	5.3	4.0	2.5	2.3
DLXb	23	5	> 1600	> 16800	> 1000	> 1000	> 1000	535.0
Bakery-2	10	4	59.0	383	9.1	7.1	2.9	2.7
BRP	22	9	561.2	5040	190.2	221.0	25.4	39.9
OOO	8	2	*	*	33.0	-	> 1000	> 1000

**Fig. 2. Hardware example results.** “Iter” is the number of iteration for abstract reachability analysis, “SVC Time” is the time spent inside SVC decision procedure, “# of Calls” denote the number of calls to SVC, “FI” and “SEP” denote the Boolean Encoding using finite-instantiation and separation predicates respectively. A “\*” indicates that SVC produces spurious answer due to rational interpretation. A “-” indicates an unexpected core dump with separation predicate encoding.

Example	# of Predicates	Explicit		SATMC			
		Calls	Time	SEP		FI	
				# Prop-vars	Time (sec)	# Prop-vars	Time (sec)
sl.0	12	955	129.5	64	3.8	60	2.6
sl.20	10	631	81.5	77	4.9	74	3.0
sl.56	11	821	110.2	65	5.4	54	2.6
sl.65	10	469	57.2	50	3.0	46	1.7
dr.10	19	>7576	>1000	162	9.2	115	9.9
dr.13	20	>7351	>1000	234	44.7	161	35.3
dr.14	20	>7189	>1000	232	103.5	157	25.6
dr.15	23	>7237	>1000	336	68.2	198	700.4
dr.16	13	2023	172.2	96	10.8	129	30.6
dr.17	15	3041	507	82	5.4	105	6.1
dr.18	18	>7099	>1000	153	130.6	180	49.7
dr.3	13	2023	355	100	9.0	125	7.0
dr.6	13	3355	596	96	8.1	129	7.8

**Fig. 3. Results over SLAM formulas.** “Calls” denotes the number of calls to the decision procedure SVC, “Prop-vars” denotes the number of propositional variables in the final propositional encoding. “SEP” denotes the method of encoding using separation predicates, and “FI” denotes the encoding using finite-instantiation. For the explicit version, the process was stopped after 1000s spent in the decision procedure.

**Software Benchmarks** For software benchmarks, we generated several problem instances from the SLAM [1] toolkit for Microsoft Windows device-driver verification. For each of these examples, the expression for the weakest precondition ( $WP$ ) and the set of predicates are supplied. The tool computes the *Weakest Boolean Precondition* ( $WPB$ ) (as described in Section 3). We have run more than 100 such examples. In Fig 3, we report the results on some of the benchmarks with greater than 10 predicates. The symbolic method also outperforms the explicit method on smaller set of predicates. On all these examples, the SATMC based solver outperformed the NuSMV based method. SATMC solver took at most a few seconds to solve most examples, whereas NuSMV took several minutes to solve the bigger examples. This is primarily due to the large number of Boolean variables in the final encoding.

## 5.1 Discussion

We have found that the BDD-based approach is more sensitive to the number of variables in the final encoding rather than the number of predicates. This is because the size of the intermediate BDD depends on the sum of the number of quantified and unquantified variables. This makes it useful for applications where the number of predicates are large, but the Boolean encoding has a smaller number of Boolean variables. This is evident in the example with Semaphore, DLXa, DLXb, Bakery-2 etc. where the NuSMV method outperforms the SATMC-based method. The SATMC based approach is more robust in the presence of large number of bound variables. This is evident in the case of the software verification benchmarks, where the number of Boolean variables is in excess of 100.

The large number of calls to the decision procedure for the DLXb example can be explained as follows. For this model, the set of reachable states is extremely dense and results in a dense abstract state space too. Therefore, the number of cubes to enumerate is very large. This is one reason why even the SATMC based approach takes a long time to solve the example.

For parameterized systems like the out-of-order processor or cache-coherence protocols, we have found the SATMC based method to outperform the NuSMV-based methods on most examples. This is because, even though the number of predicates is small (typically 10–15), the instantiation of quantifiers produces a lot of terms in the first-order formula, which translates to a large number of Boolean variables (almost 500) in the final formula.

**Acknowledgements** The first author is grateful to Pankaj Chauhan for his help in understanding NuSMV and his suggestions. We are also grateful to Daniel Kroening for letting us use SATMC, and spending time to create an interface to our tool. We are also grateful to Sanjit Seshia and Amit Goel for their comments and help with the tools.

## References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, November 1996.
3. C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer-Aided Verification (CAV '02)*, LNCS 2404, 2002.
4. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), August 1986.
5. R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.

6. 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 *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
7. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, June 1994.
8. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *Computer-Aided Verification (CAV'99)*, LNCS 1633, July 1999.
9. E. Clarke, D. Kroening, and P. Chauhan. Fixpoint computation for circuits using Symbolic Simulation and SAT. In *Preparation*, 2003.
10. D. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proceedings of Fourth Annual Symposium on Principles of Programming Languages (POPL '77)*, 1977.
11. S. Das and D. Dill. Successive approximation of abstract transition relations. In *IEEE Symposium of Logic in Computer Science (LICS '01)*, June 2001.
12. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.
13. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '02)*, 2002.
14. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer-Aided Verification (CAV '94)*, LNCS 818, June 1994.
15. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer-Aided Verification (CAV '97)*, LNCS 1254, June 1997.
16. S. K. Lahiri, C. Pixley, and K. Albin. Experience with term level modeling and verification of the MCORE microprocessor core. In *Proc. IEEE High Level Design Validation and Test (HLDVT 2001)*, November 2001.
17. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, November 2002.
18. K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In *Computer Aided Verification, (CAV '02)*, LNCS 2404, 2002.
19. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.
20. K. Namjoshi and R. Kurshan. Syntactic program transformations for automatic abstraction. In *Computer-Aided Verification (CAV 2000)*, LNCS 1855, July 2000.
21. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, 2001.
22. H. Saidi and N. Shankar. Abstract and Model Check while you Prove. In *Computer-Aided Verification (CAV '99)*, LNCS 1633, July 1999.
23. O. Strichmann. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, November 2002.
24. O. Strichmann, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with sat. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
25. M. N. Velev and R. E. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions and Branch Predication. In *37th Design Automation Conference (DAC '00)*, June 2000.

# Unbounded, Fully Symbolic Model Checking of Timed Automata Using Boolean Methods

Sanjit A. Seshia and Randal E. Bryant

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA  
{Sanjit.Seshia, Randy.Bryant}@cs.cmu.edu

**Abstract.** We present a new approach to unbounded, fully symbolic model checking of timed automata that is based on an efficient translation of quantified separation logic to quantified Boolean logic. Our technique preserves the interpretation of clocks over the reals and can check any property in timed computation tree logic. The core operations of eliminating quantifiers over real variables and deciding the validity of separation logic formulas are respectively translated to eliminating quantifiers on Boolean variables and checking Boolean satisfiability (SAT). We can thus leverage well-known techniques for Boolean formulas, including Binary Decision Diagrams (BDDs) and recent advances in SAT and SAT-based quantifier elimination. We present preliminary empirical results for a BDD-based implementation of our method.

## 1 Introduction

*Timed automata* [2] have proved to be a useful formalism for modeling real-time systems. A timed automaton is a generalization of a finite automaton with a set of real-valued clock variables. The state space of a timed automaton thus has a finite component (over Boolean state variables) and an infinite component (over clock variables). Several model checking techniques for timed automata have been proposed over the past decade. These can be classified, on the one hand, as being either *symbolic* or *fully symbolic*, and on the other, as being *bounded* or *unbounded*. Symbolic techniques use a symbolic representation for the infinite component of the state space, and either symbolic or explicit representations for the finite component. In contrast, fully symbolic methods employ a single symbolic representation for both finite and infinite components of the state space. Bounded model checking techniques work by unfolding the transition relation  $d$  times, finding counterexamples of length up to  $d$ , if they exist. As in the untimed case, these methods suffer from the limitation that, unless a bound on the length of counterexamples is known, they cannot verify the property of interest. Unbounded methods, on the other hand, can produce a guarantee of correctness.

The theoretical foundation for unbounded, fully symbolic model checking of timed automata was laid by Henzinger et al. [9]. The characteristic function of a set of states is a formula in *Separation Logic*, a quantifier-free fragment of first-order logic. Formulas in Separation Logic (SL) are Boolean combinations of

Boolean variables and predicates of the form  $x_i \bowtie x_j + c$  where  $\bowtie \in \{>, \geq\}$ ,  $x_i$  and  $x_j$  are real-valued variables, and  $c$  is a constant. *Quantified Separation Logic* (QSL) is an extension of SL with quantifiers over real and Boolean variables. The most important model checking operations involve deciding the validity of SL formulas and eliminating quantifiers on real variables from QSL formulas.

In this paper, we present the first approach to unbounded, fully symbolic model checking of timed automata that is based on a Boolean encoding of SL formulas and that preserves the interpretation of clocks over the reals. Unlike many other fully symbolic techniques, our method can be used to model check any property in Timed Computation Tree Logic (TCTL), a generalization of CTL. The main theoretical contribution of this paper is a new technique for transforming the problem of eliminating quantifiers on real variables to one of eliminating quantifiers on Boolean variables. In some cases, we can avoid introducing Boolean quantification altogether. These techniques, in conjunction with previous work on deciding SL formulas via a translation to Boolean satisfiability (SAT) [16], allow us to leverage well-known techniques for manipulating quantified Boolean formulas, including Binary Decision Diagrams (BDDs) and recent work on SAT and SAT-based quantifier elimination [11].

**Related Work.** The work that is most closely related to ours is the approach based on representing SL formulas using Difference Decision Diagrams (DDD) [12]. A DDD is a BDD-like data structure, where the node labels are generalized to be separation predicates rather than just Boolean variables, with the ordering of predicates induced by an ordering of clock variables. This predicate ordering permits the use of local reduction operations, such as eliminating inconsistent combinations of two predicates that involve the same pair of clock variables. Deciding a SL formula represented as a DDD is done by eliminating all inconsistent paths in the DDD. This is done by enumerating all paths in the DDD and checking the satisfiability of the conjunction of predicates on each path using a constraint solver based on the Bellman-Ford shortest path algorithm. Note that each path can be viewed as a disjunct in the Disjunctive Normal Form (DNF) representation of the DDD, and in the worst case there can be exponentially many calls to the constraint solver. Quantifier elimination is performed by the Fourier-Motzkin technique [8], which also requires enumerating all possible paths. In contrast, our Boolean encoding method is general in that any representation of Boolean functions may be used. Our decision procedure and quantifier elimination scheme use a direct translation to SAT and Boolean quantification respectively, avoiding the need to explicitly enumerate each DNF term. In theory, the use of DDDs permits unbounded, fully symbolic model checking of TCTL; however, the DDD-based model checker [12] can only check reachability properties (these can express safety and bounded-liveness properties [1]).

UPPAAL2K and KRONOS are unbounded, symbolic model checkers that explicitly enumerate the discrete component of the state space. KRONOS uses Difference Bound Matrices (DBMs) as the symbolic representation [18] of the infinite component. UPPAAL2K uses, in addition, Clock Difference Diagrams (CDDs)

to symbolically represent unions of convex clock regions [4]. In a CDD, a node is labeled by the difference of a pair of clock variables, and each outgoing edge from a node is labeled with an interval bounding that difference. While KRONOS can check arbitrary TCTL formulas, UPPAAL2K is limited to checking reachability properties and very restricted liveness properties such as the CTL formula **AFp**.

RED is an unbounded, fully symbolic model checker based on a data structure called the Clock Restriction Diagram (CRD) [17]. The CRD is similar to a CDD, labeling each node with the difference between two clock variables. However, each outgoing edge from a node is labeled with an upper bound, instead of an interval. RED represents separation formulas by a combined BDD-CRD structure, and can model check TCTL formulas.

A fully symbolic version of KRONOS using BDDs has been developed by interpreting clock variables over integers [6]; however, this approach is restricted to checking reachability for the subclass of closed timed automata<sup>1</sup>, and the encoding blows up with the size of the integer constants. Rabbit [5] is a tool based on this approach that additionally exploits compositional methods to find good BDD variable orderings. In comparison, our technique applies to all timed automata and its efficiency is far less sensitive to the size of constants. Also, the variable ordering methods used in Rabbit could be used in a BDD-based implementation of our technique.

Many fully symbolic, but bounded model checking methods based on SAT have been developed recently (e.g., [3,13]). These algorithms cannot be directly extended to perform unbounded model checking.

The rest of the paper is organized as follows. We define notation and present background material in Section 2. We describe our new contributions in Sections 3 and 4. We conclude in Section 5 with experimental results and ongoing work. Details including proofs of theorems stated in the paper can be found in our technical report [14].

## 2 Background

We begin with a brief presentation of background material, based on papers by Alur [2] and Henzinger et al. [9]. We refer the reader to these papers for details.

### 2.1 Separation Logic

*Separation logic* (SL), also known as *difference logic*, is a quantifier-free fragment of first-order logic. A formula  $\phi$  in separation logic is a Boolean combination of Boolean variables and *separation predicates* (also known as *difference bound constraints*) involving real-valued variables, as given by the following grammar:

$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid b \mid \neg\phi \mid \phi \wedge \phi \mid x_i \geq x_j + c \mid x_i > x_j + c$$

We use a special variable  $x_0$  to denote the constant 0; this allows us to express bounds of the form  $x \geq c$ . We will however use both  $x \bowtie c$  and  $x \bowtie x_0 + c$ , where

<sup>1</sup> Clock constraints in a closed timed automaton do not contain strict inequalities.



$\bowtie \in \{>, \geq\}$ , as suits the context. We will denote Boolean variables by  $b, b_1, b_2, \dots$ , real variables by  $x, x_1, x_2, \dots$ , and SL formulas by  $\phi, \phi_1, \phi_2, \dots$ . Note that the relations  $>$  and  $\geq$  suffice to represent equalities and other inequalities.

Characteristic functions of sets of states of timed automata are SL formulas. Deciding the satisfiability of a SL formula is NP-complete [9].

**Quantified Separation Logic.** Separation logic can be generalized by the addition of quantifiers over both Boolean and real variables. This yields *quantified separation logic* (QSL). The satisfiability problem for QSL is PSPACE-complete [10]. We will denote QSL formulas by  $\omega, \omega_1, \omega_2, \dots$ .

## 2.2 Timed Automata

A *timed automaton*  $\mathcal{T}$  is a tuple  $\langle \mathcal{L}, \mathcal{L}_0, \Sigma, \mathcal{X}, \mathcal{I}, \mathcal{E} \rangle$ , where  $\mathcal{L}$  is a finite set of locations,  $\mathcal{L}_0 \subseteq \mathcal{L}$  is a finite set of initial locations,  $\Sigma$  is a finite set of labels used for product construction,  $\mathcal{X}$  is a finite set of non-negative real-valued clock variables,  $\mathcal{I}$  is a function mapping a location to a SL formula (called a *location invariant*), and  $\mathcal{E}$  is the transition relation, a subset of  $\mathcal{L} \times \Psi \times \Sigma \times \mathcal{R} \times \mathcal{L}$ , where  $\Psi$  is a set of SL formulas that form enabling *guard* conditions for each transition, and  $\mathcal{R}$  is a set of *clock reset assignments*. A location invariant is the condition under which the system can stay in that location. A clock reset assignment is of the form  $x_i := x_0 + c$  or  $x_i := x_j$ , where  $x_i, x_j \in \mathcal{X}$  and  $c$  is a non-negative rational constant,<sup>2</sup> and indicates that the clock variable on the left-hand side of the assignment is reset to the value of the expression on the right-hand side. We will denote guards by  $\psi, \psi_1, \dots$ . The invariant  $\mathcal{I}_{\mathcal{T}}$  for the timed automaton  $\mathcal{T}$  is defined as  $\mathcal{I}_{\mathcal{T}} = \bigwedge_{l \in \mathcal{L}} [\text{enc}(l) \implies \mathcal{I}(l)]$ , where  $\text{enc}(l)$  denotes the Boolean encoding of location  $l$ . We will also denote a transition  $t \in \mathcal{E}$  as  $\psi \implies A$ , where  $\psi$  is a guard condition over both Boolean state variables (used to encode locations) and clock variables of the system, and  $A$  is a set of assignments to clock and Boolean state variables.

Two timed automata are composed by synchronizing over common labels. We refer the reader to Alur's paper [2] for a formal definition of product construction. Note that in contrast to the definition of timed automata given by Alur [2], we allow location invariants and guards to be arbitrary SL formulas, rather than simply conjunctions over separation predicates involving clock variables.

## 2.3 Fully Symbolic Model Checking

Properties of timed automata can be expressed in a generalization of the  $\mu$  calculus called the *timed  $\mu$*  ( $\mathbf{T}\mu$ ) calculus. Henzinger et al. [9] showed that the  $\mathbf{T}\mu$  calculus can express TCTL, the dense-real-time version of CTL. They have given a fully symbolic model checking algorithm that verifies if a timed automaton  $\mathcal{T}$  satisfies a specification given as a  $\mathbf{T}\mu$  formula  $\varphi$ . The algorithm terminates, generating a SL formula  $|\varphi|$ , such that, if  $\mathcal{T}$  is *non-zeno* (i.e., time can diverge

<sup>2</sup> The assignment  $x_i := c$  is represented as  $x_i := x_0 + c$ . Wherever we use  $x_i$  to denote a clock variable,  $i > 0$ .

from any state), then  $|\varphi|$  is equivalent to  $\mathcal{I}_{\mathcal{T}}$  iff  $\mathcal{T}$  satisfies  $\varphi$ . For lack of space, we omit details of the  $\mathbf{T}\mu$  calculus, TCTL, and the model checking algorithm; these can be found in our technical report [14] and in the original paper [9].

Our work is based on Henzinger et al.'s model checking algorithm. It performs backward exploration of the state space and relies on three fundamental operations:

1. **Time Elapse:**  $\phi_1 \rightsquigarrow \phi_2$  denotes the set of all states that can reach the state set  $\phi_2$  by allowing time to elapse, while staying in state set  $\phi_1$  at all times in between. Formally,

$$\phi_1 \rightsquigarrow \phi_2 \doteq \exists \delta \{ \delta \geq x_0 \wedge \phi_2 + \delta \wedge \forall \epsilon [x_0 \leq \epsilon \leq \delta \implies \phi_1 + \epsilon] \} \quad (1)$$

where  $\phi + \delta$  denotes the formula obtained by adding  $\delta$  to all clock variables occurring in  $\phi$ , computed as  $\phi[x_i + \delta/x_i, 1 \leq i \leq n]$ , where  $x_1, x_2, \dots, x_n$  are the clock variables in  $\phi_i$  (i.e., not including the zero variable  $x_0$ ).

Consider the formula on the right hand side of Equation 1. This formula is not in QSL, since it includes expressions that are the sum of two real variables (e.g.,  $x + \delta$ ). However, it can be transformed to a QSL formula, by using instead of  $\delta$  and  $\epsilon$ , variables  $\bar{\delta}$  and  $\bar{\epsilon}$  that represent their negations:

$$\exists \bar{\delta} \{ \bar{\delta} \leq x_0 \wedge \phi_2 + (-\bar{\delta}) \wedge \forall \bar{\epsilon} [ \bar{\delta} \leq \bar{\epsilon} \leq x_0 \implies \phi_1 + (-\bar{\epsilon}) ] \} \quad (2)$$

Formula 2 is expressible in QSL, since the substitution  $\phi[x_i + (-\bar{\delta})/x_i, 1 \leq i \leq n]$  can be computed as  $\phi[\bar{\delta}/x_0]$ .<sup>3</sup> This yields,

$$\exists \bar{\delta} \{ \bar{\delta} \leq x_0 \wedge \phi_2[\bar{\delta}/x_0] \wedge \forall \bar{\epsilon} ( \bar{\delta} \leq \bar{\epsilon} \leq x_0 \implies \phi_1[\bar{\epsilon}/x_0] ) \} \quad (3)$$

Finally, we can rewrite Formula 3 purely in terms of existential quantifiers:

$$\exists \bar{\delta} \{ \bar{\delta} \leq x_0 \wedge \phi_2[\bar{\delta}/x_0] \wedge \neg \exists \bar{\epsilon} ( \bar{\epsilon} \leq x_0 \wedge \bar{\delta} \leq \bar{\epsilon} \wedge \neg \phi_1[\bar{\epsilon}/x_0] ) \} \quad (4)$$

A procedure for performing the time elapse operation therefore requires one for eliminating (existential) quantifiers over real variables from a SL formula.

2. **Assignment:**  $\phi[A]$ , where  $A$  is a set of assignments, denotes the formula obtained by simultaneously substituting in  $\phi$  the right hand side of each assignment in  $A$  for the left hand side. Formally, if  $A$  is the list  $b_1 := \phi_1, \dots, b_k := \phi_k, x_1 := x_{j_1} + c_1, \dots, x_n := x_{j_n} + c_n$ , where each  $b_i$  is a Boolean variable, each  $x_j$  is a clock variable, and for each  $x_{j_l}, j_l = 0$  or  $c_l = 0$ , then

$$\phi[A] = \phi[\phi_1/b_1, \dots, \phi_k/b_k, x_{j_1} + c_1/x_1, \dots, x_{j_n} + c_n/x_n]$$

Assignments are thus performed via substitutions of variables.

3. **Checking Termination:** The termination condition of the fixpoint iteration in the model checking algorithm checks if one set of states,  $\phi_{new}$ , is contained in another,  $\phi_{old}$ . This check is performed by deciding the validity of the SL formula  $\phi \doteq \phi_{new} \implies \phi_{old}$  (or equivalently, the satisfiability of  $\neg\phi$ ).

---

<sup>3</sup> Note that substituting  $x_0$  by  $\bar{\delta}$  or  $\bar{\epsilon}$  can be viewed as shifting the zero reference point to a more negative value, thus increasing the value of any clock variable relative to zero (as observed, e.g., in [3,12]).

### 3 Model Checking Operations Using Boolean Encoding

We now show how to implement the fundamental model checking operations using a Boolean encoding of separation predicates. We first describe how our encoding allows us to replace quantification of real variables by quantification of Boolean variables. This builds on previous work on deciding a SL formula by transformation to a Boolean formula [16]. We then show how we represent SL formulas as Boolean formulas, allowing the model checking operations to be implemented as operations in Quantified Boolean Logic (QBL), and enabling the use of QBL packages, e.g., a BDD package.

In the remainder of this section, we will use  $\phi$  to denote a SL formula over real variables  $x_1, x_2, \dots, x_n$ , and Boolean variables  $b_1, b_2, \dots, b_k$ . Also, let  $\bowtie, \bowtie_1, \bowtie_2 \in \{>, \geq\}$ .

#### 3.1 From Real Quantification to Boolean Quantification

Consider the QSL formula  $\omega_a \doteq \exists x_a. \phi$ , where  $a \in [1..n]$ .

We transform  $\omega_a$  to an equivalent QSL formula  $\omega_{bool}$  with quantifiers over only Boolean variables in the following three steps:

1. *Encode separation predicates:*

Consider each separation predicate in  $\phi$  of the form  $x_i \bowtie x_j + c$  where either  $i = a$  or  $j = a$ . For each such predicate, we generate a corresponding Boolean variable  $e_{i,j}^{\bowtie,c}$ . Separation predicates that are negations of each other are represented by Boolean literals (true or complemented variables) that are negations of each other; however, for ease of presentation, we will extend the naming convention for Boolean variables to Boolean literals, writing  $e_{j,i}^{>,-c}$  for the negation of  $e_{i,j}^{\geq,c}$ .

Let the added Boolean variables be  $e_{i_1,a}^{\bowtie_{i_1},c_{i_1}}, e_{i_2,a}^{\bowtie_{i_2},c_{i_2}}, \dots, e_{i_m,a}^{\bowtie_{i_m},c_{i_m}}$  for the upper bounds on  $x_a$ , and  $e_{a,j_1}^{\bowtie_{j_1},c_{j_1}}, e_{a,j_2}^{\bowtie_{j_2},c_{j_2}}, \dots, e_{a,j_{m'}}^{\bowtie_{j_{m'}},c_{j_{m'}}}$  for the lower bounds on it.

We replace each predicate  $x_a \bowtie x_j + c$  (or  $x_i \bowtie x_a + c$ ) in  $\phi$  by the corresponding Boolean variable  $e_{a,j}^{\bowtie,c}$  (or  $e_{i,a}^{\bowtie,c}$ ). Let the resulting SL formula be  $\phi_{bool}^a$ .

2. *Add transitivity constraints:*

Notice that there can be assignments to the  $e_{i,a}^{\bowtie,c}$  and  $e_{a,j}^{\bowtie,c}$  variables that have no corresponding assignment to the real valued variables. To disallow such assignments, we place constraints on these added Boolean variables. Each constraint is generated from two Boolean literals that encode predicates containing  $x_a$ . We will refer to these constraints as *transitivity constraints* for  $x_a$ .

A transitivity constraint for  $x_a$  has one of the following types:

- (a)  $e_{i,a}^{\bowtie_1,c_1} \wedge e_{a,j}^{\bowtie_2,c_2} \implies (x_i \bowtie x_j + c_1 + c_2)$ ,  
where if  $\bowtie_1 = \bowtie_2$ , then  $\bowtie = \bowtie_1$ , otherwise, we must duplicate this constraint for both  $\bowtie = \bowtie_1$  and for  $\bowtie = \bowtie_2$ .
- (b)  $e_{i,j}^{\bowtie_1,c_1} \implies e_{i,j}^{\bowtie_2,c_2}$ , where  $c_1 > c_2$  and either  $i = a$  or  $j = a$ .

(c)  $e_{i,j}^{>,c} \implies e_{i,j}^{\geq,c}$ , where either  $i = a$  or  $j = a$ .

Note that a constraint of type (a) involves a separation predicate ( $x_i \bowtie x_j + c_1 + c_2$ ). This predicate might not be present in the original formula  $\phi$ .<sup>4</sup> After generating all transitivity constraints for  $x_a$ , we conjoin them to get the SL formula  $\phi_{cons}^a$ .

3. Finally, generate the QSL formula  $\omega_{bool}$  given below:

$$\exists e_{i_1,a}^{\bowtie_{i_1,c_{i_1}}}, e_{i_2,a}^{\bowtie_{i_2,c_{i_2}}}, \dots, e_{i_m,a}^{\bowtie_{i_m,c_{i_m}}} . \exists e_{a,j_1}^{\bowtie_{j_1,c_{j_1}}}, e_{a,j_2}^{\bowtie_{j_2,c_{j_2}}}, \dots, e_{a,j_{m'}}^{\bowtie_{j_{m'},c_{j_{m'}}}} . [\phi_{cons}^a \wedge \phi_{bool}^a]$$

We formalize the correctness of this transformation in the following theorem.

**Theorem 1.**  $\omega_a$  and  $\omega_{bool}$  are equivalent.

*Example 1.* Let  $\omega_a = \exists x_a . \phi$  where  $\phi = x_a \leq x_0 \wedge x_1 \geq x_a \wedge x_2 \leq x_a$ . Then,  $\phi_{bool}^a = e_{0,a}^{\geq,0} \wedge e_{1,a}^{\geq,0} \wedge e_{a,2}^{\geq,0}$ .  $\phi_{cons}^a$  is the conjunction of the following constraints:

1.  $e_{0,a}^{\geq,0} \wedge e_{a,2}^{\geq,0} \implies x_0 \geq x_2$
2.  $e_{1,a}^{\geq,0} \wedge e_{a,2}^{\geq,0} \implies x_1 \geq x_2$

Then,  $\omega_{bool} = \exists e_{0,a}^{\geq,0}, e_{1,a}^{\geq,0}, e_{a,2}^{\geq,0} . [\phi_{cons}^a \wedge \phi_{bool}^a]$  evaluates to  $x_0 \geq x_2 \wedge x_1 \geq x_2$ .  $\square$

The quantifier transformation procedure described here works even when  $\phi$  is replaced by a QSL formula with quantifiers only over Boolean variables.

### 3.2 Deciding SL Formulas

Suppose we want to decide the satisfiability of  $\phi$ . Note that  $\phi$  is satisfiable iff the QSL formula  $\omega_{1..n} = \exists x_1, x_2, \dots, x_n . \phi$  is.

Using Theorem 1, we can transform  $\omega_{1..n}$  to an equivalent QSL formula  $\omega_{bool}$  with existential quantifiers only over Boolean variables encoding all separation predicates. As  $\omega_{bool}$  is a QBL formula with only existential quantifiers, we can simply discard the quantifiers and use a Boolean satisfiability checker to decide the resulting Boolean formula.

Note that the transformation described above can be viewed as one way to implement the procedure of Strichman et al. [16].

### 3.3 Representing SL Formulas as Boolean Formulas

In our presentation up to this point, we have not used any specific representation of SL formulas. In practice, we encode a SL formula  $\phi$  as a Boolean formula  $\beta$ . The encoding is performed as follows. Consider each separation predicate  $x_i \bowtie x_j + c$  in  $\phi$ . As in Section 3.1 earlier, we introduce a Boolean variable  $e_{i,j}^{\bowtie,c}$  for  $x_i \bowtie x_j + c$ , only this time we do it for every single separation predicate. Also as before, separation predicates that are negations of each other are represented

<sup>4</sup> This addition is analogous to the “tightening” step performed in difference-bound matrix based algorithms

by Boolean literals that are negations of each other. We then replace each separation predicate in  $\phi$  by its corresponding Boolean literal. The resulting Boolean formula is  $\beta$ .

Clearly,  $\beta$ , by itself, stores insufficient information for generating transitivity constraints. Therefore, we also store the 1-1 mapping of separation predicates to the Boolean literals that encode them. However, this mapping is used only lazily, i.e., when generating transitivity constraints during quantification and in deciding SL formulas.

**Substitution.** Given the representation described above, we can implement substitution of a clock variable as follows. For a clock variable  $x_i$ , we perform the substitution  $[x_i \leftarrow x_k + d]$  (where  $k = 0$  or  $d = 0$ ), by replacing all Boolean variables of the form  $e_{i,j}^{\bowtie,c}$  and  $e_{j,i}^{\bowtie',c'}$ , for all  $j$ , by variables  $e_{k,j}^{\bowtie,c-d}$  and  $e_{j,k}^{\bowtie',c'+d}$  respectively, creating fresh replacement variables if necessary. Substitution of a Boolean state variable by the Boolean encoding of a separation formula is done by Boolean function composition.

## 4 Optimizations

The methods presented in Section 3 can be optimized in a few ways. First, we can be more selective in deciding when to add transitivity constraints. Second, we can compute the time elapse operator more efficiently by a method that does not explicitly introduce the bound real variable  $\bar{e}$ , and hence does not introduce new quantifiers over Boolean variables. The final optimization concerns eliminating paths in a BDD representation that violate transitivity constraints. As is well-known, the size of a BDD is very sensitive to the number and ordering of BDD variables. In the case of model checking timed automata, new Boolean variables are created as the model checking proceeds, while generating transitivity constraints, and while performing substitutions of clock variables. This has the potential to add several BDD variables on each iteration. While all three techniques presented in this section help in reducing the number of BDD variables, only the last technique is specialized for BDDs.

### 4.1 Determining if Bounds Are Conjoined

Suppose  $\phi$  is a SL formula with Boolean encoding  $\beta$ , and we wish to eliminate the quantifier in  $\exists x_a.\phi$ . As described in Section 3.1, a transitivity constraint for  $x_a$  involves two Boolean literals that encode separation predicates involving  $x_a$ . For a syntactic representation of  $\beta$ , as the number of constraints grows, so does the size of  $[\beta_{cons}^a \wedge \beta_{bool}^a]$ , the Boolean encoding of  $[\phi_{cons}^a \wedge \phi_{bool}^a]$ . Further, new separation predicates can be added when a transitivity constraint is generated from an upper bound and a lower bound on  $x_a$ . For a BDD-based implementation, this corresponds to the addition of a new BDD variable. We would therefore like to avoid adding transitivity constraints wherever possible.

In fact, we only need to add a constraint involving an upper bound literal and a lower bound literal if they are conjoined in a minimized DNF representation of  $\beta$ .<sup>5</sup> From a geometric viewpoint, this means that we check that the predicates corresponding to the two literals are bounds for the same convex clock region. This check can be posed as a Boolean satisfiability problem, which is easily solved using a BDD representation of  $\beta$ . Let the literals be  $e_1$  and  $e_2$ . Then, we use cofactoring and Boolean operations to compute the following Boolean formula:

$$e_1 \wedge e_2 \wedge [\beta|_{e_1=\text{true}} \wedge \neg(\beta|_{e_1=\text{false}})] \wedge [\beta|_{e_2=\text{true}} \wedge \neg(\beta|_{e_2=\text{false}})] \quad (5)$$

Formula 5 expresses the Boolean function corresponding to the disjunction of all terms in the minimized DNF representation of  $\beta$  that contain both  $e_1$  and  $e_2$  in true form. Therefore, if Formula 5 is satisfiable, it means that  $e_1$  and  $e_2$  are conjoined, and we must add a transitivity constraint involving them both.

Note however, that since  $\beta$  does not, by itself, represent the original SL formula  $\phi$ , finding that  $e_1$  and  $e_2$  are conjoined in  $\beta$  does not imply that they are bounds in the same convex region of  $\phi$ . However, the converse is true, so our method is sound.

## 4.2 Quantifier Elimination by Eliminating Upper Bounds on $x_0$

The definition of the time elapse operation introduces two quantified non-clock real variables:  $\bar{\delta}$  and  $\bar{\epsilon}$ . We can exploit the special structure of the QSL formula for the time elapse operation so as to avoid introducing  $\bar{\epsilon}$  altogether. Thus, we can avoid adding new quantified Boolean variables encoding predicates involving  $\bar{\epsilon}$ .

Consider the inner existentially quantified SL formula in Formula 4 in Section 2.3, reproduced here:

$$\exists \bar{\epsilon} (\bar{\epsilon} \leq x_0 \wedge \bar{\delta} \leq \bar{\epsilon} \wedge \neg \phi_1[\bar{\epsilon}/x_0])$$

Grouping the inequality  $\bar{\delta} \leq \bar{\epsilon}$  with the formula  $\neg \phi_1[\bar{\epsilon}/x_0]$ , we get:

$$\exists \bar{\epsilon} \{ \bar{\epsilon} \leq x_0 \wedge (\bar{\delta} \leq x_0 \wedge \neg \phi_1)[\bar{\epsilon}/x_0] \} \quad (6)$$

Finally, treating  $\bar{\delta}$  as a clock variable, we can revert back to  $\epsilon$  from  $\bar{\epsilon}$ , transforming Formula 6 to the following form:

$$\exists \epsilon [\epsilon \geq x_0 \wedge (\bar{\delta} \leq x_0 \wedge \neg \phi_1) + \epsilon] \quad (7)$$

Formula 7 is a special case of the formula  $\omega_\epsilon$  given by

$$\omega_\epsilon = \exists \epsilon. \epsilon \geq x_0 \wedge \phi + \epsilon$$

for some SL formula  $\phi$ . From a geometric viewpoint,  $\phi$  is a region in  $\mathbb{R}^n$  and  $\omega_\epsilon$  is the shadow of  $\phi$  for a light source at  $\infty^n$ . Examples of  $\phi$  and the corresponding  $\omega_\epsilon$  are shown in Figures 1(a) and 1(c) respectively.

We can transform  $\omega_\epsilon$  to an equivalent SL formula  $\phi_{ub}$  by eliminating upper bounds on  $x_0$ , i.e., Boolean variables of the form  $e_{i,0}^{\leq, c}$ . The transformation is performed iteratively in the following steps:

<sup>5</sup> A conservative, syntactic variant of this idea was earlier proposed by Strichman [15].

1. Let  $\phi_0 = \phi$ . Let  $e_{i_1,0}^{\bowtie_1,c_1}, e_{i_2,0}^{\bowtie_2,c_2}, \dots, e_{i_m,0}^{\bowtie_m,c_m}$  be Boolean literals encoding all upper bounds on  $x_0$  that occur in  $\phi$ .
2. For  $j = 1, 2, \dots, m$ , we construct  $\phi_j$  as follows:
  - (a) Replace all occurrences of  $x_{i_j} \bowtie_j x_0 + c_j$  in  $\phi_{j-1}$  with  $e_{i_j,0}^{\bowtie_j,c_j}$  to get  $\phi_{bool}^{0,j-1}$ .
  - (b) Construct  $\phi_{cons}^{0,j-1}$ , the conjunction of all transitivity constraints<sup>6</sup> for  $x_0$  involving  $e_{i_j,0}^{\bowtie_j,c_j}$  and clock variables in  $\phi_{bool}^{0,j-1}$ .
  - (c) Construct the formula  $\phi_j$ , a disjunction of two terms:

$$\phi_j = \{(\phi_{bool}^{0,j-1} \wedge \phi_{cons}^{0,j-1})|_{e_{i_j,0}^{\bowtie_j,c_j}=\text{true}}\} \vee \{[\neg(x_{i_j} \bowtie_j x_0 + c_j)] \wedge [\phi_{bool}^{0,j-1}]_{e_{i_j,0}^{\bowtie_j,c_j}=\text{false}}\}$$

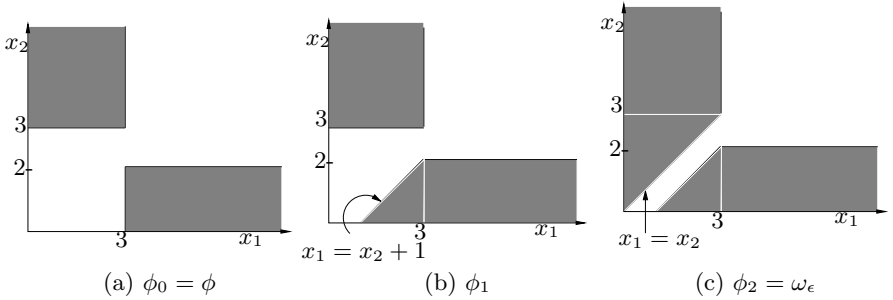
The first disjunct is the region obtained by dropping the bound  $x_{i_j} \bowtie_j x_0 + c_j$  from convex sub-regions of  $\phi_{j-1}$  where it is a lower bound on  $x_{i_j}$ , while letting time elapse backward. The second disjunct corresponds to sub-regions where  $\neg(x_{i_j} \bowtie_j x_0 + c_j)$  is an upper bound; these regions are left unchanged.

The output of the above transformation,  $\phi_{ub}$ , is given by  $\phi_{ub} = \phi_m$ . The correctness of this procedure is formalized in the following theorem.

**Theorem 2.**  $\omega_\epsilon$  and  $\phi_{ub}$  are equivalent.

*Example 2.* Let the subformula  $\phi$  of  $\omega_\epsilon$  be

$$\phi = (x_1 \geq x_0 + 3 \wedge x_2 \leq x_0 + 2) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$



**Fig. 1.** Eliminating upper bounds on  $x_0$ . Only the positive quadrant is shown.

$\phi$  is depicted geometrically as the shaded region in Figure 1(a). It comprises two sub-regions, one for each disjunct. The lower bounds on these regions,  $x_1 \geq x_0 + 3$  and  $x_2 \geq x_0 + 3$ , are upper bounds on  $x_0$ . We encode these by  $e_{1,0}^{\geq,3}$  and  $e_{2,0}^{\geq,3}$ .

<sup>6</sup> We can use the optimization technique of Section 4.1 in this step.

Figure 1(b) shows  $\phi_1$ , the result of eliminating  $e_{1,0}^{\geq,3}$ . Formally, we calculate

$$\begin{aligned}\phi_{bool}^{0,0} &= (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \vee (\neg e_{1,0}^{\geq,3} \wedge x_2 \geq x_0 + 3) \\ \phi_{cons}^{0,0} &= (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \implies (x_1 \geq x_2 + 1)\end{aligned}$$

Then, applying step 2(c) of the transformation, we get

$$\phi_1 = (x_2 \leq x_0 + 2 \wedge x_1 \geq x_2 + 1) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$

Similarly, in the next iteration, we introduce and eliminate  $e_{2,0}^{\geq,3}$  to get  $\phi_2$ , shown in Figure 1(c), which is equivalent to  $\omega_\epsilon$ .  $\square$

Note that the QSL formula obtained after eliminating the inner quantifier in Formula 4 is not of the form  $\omega_\epsilon$ , and so we cannot avoid introducing the  $\bar{\delta}$  variable.

### 4.3 Eliminating Infeasible Paths in BDDs

Suppose  $\beta$  is the Boolean encoding of SL formula  $\phi$ . Let  $\phi_{cons}$  denote the conjunction of transitivity constraints for all real-valued variables in  $\phi$ , and let  $\beta_{cons}$  denote its Boolean encoding. Finally, denote the BDD representations of  $\beta$  and  $\beta_{cons}$  by  $\text{Bdd}(\beta)$  and  $\text{Bdd}(\beta_{cons})$  respectively.

We would like to eliminate paths in  $\text{Bdd}(\beta)$  that violate transitivity constraints, i.e., those corresponding to assignments to variables in  $\beta$  for which  $\beta_{cons} = \text{false}$ . We can do this by using the BDD **Restrict** operator, replacing  $\text{Bdd}(\beta)$  by  $\text{Restrict}(\text{Bdd}(\beta), \text{Bdd}(\beta_{cons}))$ . Informally,  $\text{Restrict}(\text{Bdd}(\beta), \text{Bdd}(\beta_{cons}))$  traverses  $\text{Bdd}(\beta)$ , eliminating a path on which  $\beta_{cons}$  is **false** as long as it doesn't involve adding new nodes to the resulting BDD. Details about the **Restrict** operator may be found in the paper by Coudert and Madre [7].

Since eliminating infeasible paths in a large BDD can be quite time consuming, we only apply this optimization to the BDD for the set of reachable states, once on each fixpoint iteration.

## 5 Experimental Results

We implemented a BDD-based model checker called TMV, that is written in OCaml and uses the CUDD package<sup>7</sup>. We have performed preliminary experiments comparing the performance of our model checker for both reachability and non-reachability TCTL properties. For reachability properties, we compare against the other unbounded, fully symbolic model checkers, viz., a DDD-based checker (DDD) [12] and RED version 4.1 [17], which have been shown to outperform UPPAAL2K and KRONOS for reachability analysis. For non-reachability properties, such as checking that a system is non-zeno, we compare against KRONOS and RED, the only other unbounded model checkers that check such properties.

<sup>7</sup> <http://vlsi.colorado.edu/~fabio/CUDD>



We ran two experiments using Fischer’s time-based mutual exclusion protocol. The first experiment compared our model checker against DDD and RED, checking that the system preserves mutual exclusion (a reachability property). In the second, we compared against KRONOS and RED for checking that the product automaton is non-zeno (a non-reachability property). All experiments were run on a notebook computer with a 1 GHz Pentium-III processor and 128 MB RAM, running Linux. We ran DDD, KRONOS, and RED with their default options. TMV used a static BDD variable ordering that is the same as the one used for the corresponding Boolean variables and separation predicates in DDD and is described in detail in our technical report [14].

Table 1(a) shows the results of the comparison against DDD and RED for checking mutual exclusion for increasing numbers of processes. For DDD and TMV, the table lists both the run-times and the peak number of nodes in the decision diagram for the reachable state set. We find that DDD outperforms TMV due to the blow-up of BDDs. In spite of the optimizations of Section 4, the peak node count in the case of DDD is less than that for TMV, since the local reduction operations performed by DDD during node creation can eliminate unnecessary DDD nodes without adding any time overhead. For example, DDD can reduce a function of the form  $e_1 \wedge e_2 \wedge e_3$  under the transitivity constraint  $[e_1 \wedge e_2] \implies e_3$  to simply the conjunction  $e_1 \wedge e_2$ . The BDD `Restrict` operator cannot always achieve this as it is sensitive to the BDD variable ordering. Furthermore, TMV contains many other BDDs, such as those for the transitivity constraints, to which we do not apply the `Restrict` optimization due to its run-time overhead. Finally, in comparison to RED, we see that while TMV is faster on the smaller benchmarks, RED’s superior memory performance enables it to complete for 7 processes while TMV runs out of memory.

**Table 1. Checking properties of Fischer’s protocol for increasing numbers of processes.** A “\*” indicates that the model checker ran out of memory.

Num. Proc.	RED		DDD		TMV	
	Time (sec.)	Time (sec.)	Reach Set (peak nodes)	Time (sec.)	Reach Set (peak nodes)	
3	0.21	0.06	130	0.11	101	
4	1.13	0.14	352	0.38	316	
5	4.53	0.33	854	1.85	1127	
6	15.11	0.90	2375	17.41	4685	
7	46.31	2.65	6346	*	*	

Num. Proc.	KRONOS		RED		TMV	
	Time (sec.)	Time (sec.)	Time (sec.)	Time (sec.)	Reach Set (peak nodes)	
3	0.03	0.28	0.24		28	
4	0.23	1.30	0.44		39	
5	1.98	5.05	0.80		54	
6	*	17.80	2.15		69	
7	*	57.95	6.61		88	

(a) Mutual Exclusion

(b) Non-Zenoness

Table 1(b) shows the comparison with KRONOS and RED for checking non-zenoness. The time for KRONOS is the sum of the times for product construction and backward model checking. We notice that while KRONOS does better for smaller numbers of processes, the product automaton it constructs grows very quickly, becoming too large to construct at 6 processes. The run times for TMV, on the other hand, grow much more gradually, demonstrating the advantages of a fully symbolic approach. For this property, the BDDs remain small even for larger numbers of processes. Thus, TMV outperforms RED, especially as the number of processes increases. These results indicate that when the representation (BDDs)

remains small, Boolean methods for quantifier elimination and deciding SL can outperform non-Boolean methods by a significant factor.

Although they are preliminary, our results indicate that our model checker based on a general purpose BDD package can outperform methods based on specialized representations of SL formulas. We are working on improving our current methods of eliminating unnecessary BDD nodes, and are also starting to work on a SAT-based implementation.

**Acknowledgments.** We thank Joël Ouaknine, Ofer Strichman, and the reviewers for useful comments. We also thank the authors of DDD and RED for providing their tools and answering our queries. This research was supported in part by a NSEG Fellowship and by ARO grant DAAD19-01-1-0485.

## References

1. L. Aceto, P. Bouyer, A. Burgueno, and K. G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 300(1-3):411-475, 2003.
2. R. Alur. Timed automata. In *Proc. CAV'99*, LNCS 1633, pages 8–22.
3. G. Audemard, A. Cimatti, A. Korniewicz, and R. Sebastiani. Bounded model checking for timed systems. In *Proc. FORTE'02*, LNCS 2529, pages 243–259.
4. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proc. CAV'99*, LNCS 1633, 341–353.
5. D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. FME'01*, LNCS 2021, pages 318–343.
6. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. CAV'97*, LNCS 1254, pages 179–190.
7. O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD'90*, pages 126–129.
8. G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory A*, 14:288–297, 1973.
9. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
10. M. Koubarakis. Complexity results for first-order theories of temporal constraints. In *Proc. KR'94*, pages 379–390.
11. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. CAV'02*, LNCS 2404, pages 250–264.
12. J. B. Møller. Simplifying fixpoint computations in verification of real-time systems. In *Proc. RTTOOLS'02* workshop.
13. P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. FTRFTT'02*, LNCS 2469.
14. S. A. Seshia and R. E. Bryant. A Boolean approach to unbounded, fully symbolic model checking of timed automata. Technical Report CMU-CS-03-117, Carnegie Mellon University, 2003.
15. O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.
16. O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Proc. CAV'02*, LNCS 2404, pages 209–222.
17. F. Wang. Efficient verification of timed automata with BDD-like data-structures. In *Proc. VMCAI'03*, LNCS 2575, pages 189–205.
18. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2):123–133, Oct. 1997.

# Digitizing Interval Duration Logic

Gaurav Chakravorty<sup>1\*</sup> and Paritosh K. Pandya<sup>2</sup>

<sup>1</sup> Indian Institute of Technology, Kanpur, India  
gchak@cse.iitk.ac.in

<sup>2</sup> Tata Institute of Fundamental Research  
Colaba, Mumbai 400005, India  
pandya@tcs.tifr.res.in

**Abstract.** In this paper, we study the verification of dense time properties by discrete time analysis. Interval Duration Logic, (IDL), is a highly expressive dense time logic for specifying properties of real-time systems. Validity checking of IDL formulae is in general undecidable. A corresponding discrete-time logic QDDC has decidable validity.

In this paper, we consider a reduction of IDL validity question to QDDC validity using notions of digitization. A new notion of Strong Closure under Inverse Digitization, SCID, is proposed. For all SCID formulae, the dense and the discrete-time validity coincide. Moreover, SCID has good algebraic properties which can be used to conveniently prove that many IDL formulae are SCID. We also give some approximation techniques to strengthen/weaken formulae to SCID form. For SCID formulae, the validity of dense-time IDL formulae can be checked using the validity checker for discrete-time logic QDDC.

## 1 Introduction

Duration Calculus (DC) is a highly expressive logic for specifying properties of real-time systems [10]. Interval Duration Logic (IDL) [8] is a variant of Duration Calculus where formulae are interpreted over *timed state sequences*. Timed state sequences [1] are a well studied model of real-time behaviour with a well-developed automata theory and tools for the analysis of such automata. In practical applications, IDL inherits much of the expressive ability of the original DC.

Model/validity checking of IDL is undecidable in general. Even for some restricted class of formulae [8], their verification requires analysis of hybrid automata, which is computationally expensive. By contrast, the Discrete-time Duration Calculus (QDDC) is decidable [7] using well developed automata theoretic techniques. A tool DCVALID permits model/validity checking of QDDC formulae for many significant examples [7].

In this paper, we investigate the reduction of IDL validity question to QDDC validity question so that QDDC tools can be used to analyse IDL formulae.

---

\* This work was partly carried out under the VSRP program of TIFR, Mumbai.

Thus, we investigate verification of dense-time properties by discrete-time analysis. There is experimental evidence that with existing techniques, automatic verification of discrete-timed systems can be significantly easier than the verification of similar dense-time systems [2]. Hence, we believe that our approach of reduction from IDL to QDDC is practically useful.

IDL models are precisely timed state sequences where states are labelled with real-valued time stamps. Denote the set of such behaviours by  $TSS_R$ . We can also interpret IDL over timed state sequences where all time stamps have only integer values. Call the set of such behaviours as  $TSS_Z$ , and IDL restricted to such behaviours as ZIDL. An IDL behaviour can be digitized to a *set* of ZIDL behaviours by approximating its time stamps to nearby integer values. We follow the notion of digitization due to Henzinger *et al* and make use of their digitization theorem which gives sufficient conditions for reducing dense-time model checking to discrete-time model checking [4].

The reduction from IDL to QDDC is carried out in two stages. In the first stage, we give a reduction from IDL to ZIDL validity which is sound only for formulae which are “Closed under Inverse Digitization” (CID), a notion proposed by Henzinger, Manna and Pnueli [4]. Unfortunately, it is quite hard to establish whether IDL formulae have CID property. Towards this we propose a new notion of “Strong Closure under Inverse Digitization” (SCID). Fortunately, SCID is preserved by most IDL operators and we are able to give a *structural characterization* of a large class of IDL formulae which are SCID. For formulae which are not SCID, we give approximations to stronger and weaker formulae which are SCID. Finally, SCID implies CID and hence for such formulae reduction from IDL to ZIDL is sound. Our logic *IDL* includes a powerful  $\frown$  (chop) operator and a notion of  $\int P$ , the accumulated amount of time for which proposition  $P$  holds in a time interval. Digitization of such properties is one of the contributions of this paper.

ZIDL is the logic of weakly monotonic integer-timed state sequences, where as QDDC is a logic of untimed state sequences. In our next reduction, we translate an arbitrary ZIDL formula  $D$  to an “equivalent” QDDC formula  $\beta(D)$ . Here, “equivalence” means preserving models under a suitable isomorphism.

The rest of the paper is organised as follows. Logic IDL is introduced in Section 2. Basic notions of digitization and closure under digitization are presented in Section 3. Digitization of IDL formulae to ZIDL formulae is presented in Section 4. Section 5 presents the reduction from ZIDL to QDDC. We illustrate our approach by a small example in Section 6, where the validity of a (dense-time) IDL formula is established using the QDDC validity checker DCVALID. The paper ends with a discussion of related work.

## 2 Interval Duration Logic

Let  $Pvar$  be the set of propositional variables (called state variables in *DC*). The set of states is  $\Sigma = 2^{Pvar}$  consisting of the set of subsets of  $Pvar$ . Let  $\mathbb{R}^0$  be the set of non-negative real numbers.

**Definition 1.** A **timed state sequence** over  $Pvar$  is a pair  $\theta = (\sigma, \tau)$  where  $\sigma = s_0 s_1 \dots s_{n-1}$  with  $s_i \in 2^{Pvar}$  is a finite non-empty sequence of states, and  $\tau = t_0 t_1 \dots t_{n-1}$  is a finite sequence of time stamps such that  $t_i \in \mathbb{R}^0$  with  $t_0 = 0$  and  $\tau$  is non-decreasing. Let  $dom(\theta) = \{0, \dots, n-1\}$  be the set of positions within the sequence  $\theta$ . Also, let the length of  $\theta$  be  $\# \theta = n$ .

Timed state sequence gives a sampled view of timed behaviour. Note that the time is *weakly monotonic* with several state changes occurring at same time [6].

The set of timed state sequences is denoted by  $TSS_R$ . We shall use  $TSS_Z$  to denote the subset of  $TSS_R$  where all time stamps have non-negative integer values.

Let  $Prop$  be the set of propositions (boolean formulae) over  $Pvar$  with 0 denoting false and 1 denoting true. The truth of proposition  $P$  can be evaluated at any position  $i$  in  $dom(\theta)$ . This is denoted by  $\theta, i \models P$ . We omit this obvious definition.

Logic  $IDL$  is a form of interval temporal logic. The set of *intervals* within a timed state sequence  $\theta$  can be defined as follows, where  $[b, e]$  denotes a pair of positions. Each interval uniquely identifies a subsequence of  $\theta$ .

$$Intv(\theta) = \{[b, e] \in dom(\theta)^2 \mid b \leq e\}$$

*Syntax of Interval Duration Logic* Let  $p, q$  range over propositional variables from  $Pvar$ , let  $P, Q$  range over propositions and  $D_1, D_2$  range over  $IDL$  formulae. Let  $c$  range over non-negative integer constants.

$[P]^0 \mid [P] \mid D_1 \frown D_2 \mid D_1 \wedge D_2 \mid \neg D \mid \eta \text{ op } c \mid \Sigma P \text{ op } c \mid \ell \text{ op } c \mid \int P \text{ op } c$  where  $\text{op} \in \{ < \mid > \mid = \mid \leq \mid \geq \}$ . Let  $IDL_l$  denote the subset of  $IDL$  formulae in which duration formulae of the form  $(\int P \text{ op } c)$  do not occur.

*Semantics of IDL* Let  $\theta, [b, e] \models D$  denote that formula  $D$  evaluates to true within a timed state sequence  $\theta$  and interval  $[b, e]$ , as defined inductively below.

$$\begin{aligned} \theta, [b, e] \models [P]^0 & \text{ iff } b = e \text{ and } \theta, b \models P \\ \theta, [b, e] \models [P] & \text{ iff } b < e \text{ and for all } m : b < m < e. \theta, m \models P \\ \theta, [b, e] \models D_1 \frown D_2 & \text{ iff for some } m : b \leq m \leq e. \\ & \theta, [b, m] \models D_1 \text{ and } \theta, [m, e] \models D_2 \\ \theta, [b, e] \models D_1 \wedge D_2 & \text{ iff } \theta, [b, e] \models D_1 \text{ and } \theta, [b, e] \models D_2 \\ \theta, [b, e] \models \neg D & \text{ iff } \theta, [b, e] \not\models D \end{aligned}$$

Now we consider the semantics of measurement formulae. Logic  $IDL$  has four different types of measurement terms:  $\eta \mid \Sigma P \mid \ell \mid \int P$ .

These represent some specific quantitative measurements over the behaviour in a given interval. We shall denote the value of a measurement term  $t$  in a timed state sequence  $\theta$  and an interval  $[b, e]$  by  $eval(t)(\theta, [b, e])$ , as defined below. Step length  $\eta$  gives the number of steps within a given interval, whereas time length  $\ell$  gives the amount of real-time spanned by a given interval. Step count  $\Sigma P$  counts the number of states for which  $P$  holds in the (left-closed-right-open) interval. Duration  $\int P$  gives amount of real-time for which proposition  $P$  holds in the

given interval. Terms  $\eta$  and  $\Sigma P$  are called *discrete measurements*, whereas terms  $\ell$  and  $\int P$  are called *dense measurements*. A measurement formula compares a measurement term with an *integer* constant.

$$\begin{aligned}
eval(\eta)(\theta, [b, e]) &= e - b, & eval(\ell)(\theta, [b, e]) &= t_e - t_b \\
eval(\Sigma P)(\theta, [b, e]) &= \sum_{i=b}^{e-1} \begin{pmatrix} 1 & \text{if } \theta, i \models P \\ 0 & \text{otherwise} \end{pmatrix} \\
eval(\int P)(\theta, [b, e]) &= \sum_{i=b}^{e-1} \begin{pmatrix} t_{i+1} - t_i & \text{if } \theta, i \models P \\ 0 & \text{otherwise} \end{pmatrix} \\
\theta, [b, e] \models t \text{ op } c &\text{ iff } eval(t)(\theta, [b, e]) \text{ op } c
\end{aligned}$$

Finally, a formula  $D$  holds for a timed state sequence  $\theta$  if it holds for the full interval spanning the whole sequence.

$$\begin{aligned}
\theta \models D &\text{ iff } \theta, [0, \#\theta - 1] \models D \\
\models D &\text{ iff } \theta \models D \text{ for all } \theta
\end{aligned}$$

### Derived Operators

- $\llbracket P \rrbracket \stackrel{\text{def}}{=} ([P]^0 \frown [P])$  and  $\llbracket P \rrbracket \stackrel{\text{def}}{=} (\llbracket P \rrbracket \frown [P]^0)$ . Formula  $\llbracket P \rrbracket$  states that proposition  $P$  holds invariantly over the interval including its end points.
- $\Diamond D \stackrel{\text{def}}{=} true \frown D \frown true$  holds provided  $D$  holds for some subinterval.
- $\Box D \stackrel{\text{def}}{=} \neg \Diamond \neg D$  holds provided  $D$  holds for all subintervals.

*Example 1.* Formula  $\Box(\llbracket P \rrbracket \Rightarrow \ell \leq 10)$  states that in any interval, if  $P$  is invariantly true then the time length of the interval is at most 10. That is,  $P$  cannot last for more than 10 time units at a stretch.

$$Follows(P, Q, d) \stackrel{\text{def}}{=} \neg \Diamond((\llbracket P \rrbracket \wedge \ell \geq d) \frown [\neg Q]^0).$$

Formula  $Follows(P, Q, d)$  states that if  $P$  has held continuously  $d$  or more time units in past, then  $Q$  must hold. Formula  $FollowsWk$  requires  $Q$  to hold only after  $P$  has held for strictly more than  $d$  time units.

$$FollowsWk(P, Q, d) \stackrel{\text{def}}{=} \neg \Diamond((\llbracket P \rrbracket \wedge \ell > d) \frown [\neg Q]^0). \quad \square$$

*Quantified Discrete-time Duration Calculus (QDDC)* is the the subset of *IDL* where dense-time measurement constructs of the form  $\ell \text{ op } c$  or  $\int P \text{ op } c$  are not used. Note that discrete time measurement constructs  $\eta \text{ op } c$  or  $\Sigma P \text{ op } c$  can still be used. For *QDDC* formulae, the time stamps  $\tau$  in behaviour  $\theta = (\sigma, \tau)$  do not play any role. Hence, we can also define the semantics of *QDDC* purely using state sequences, i.e.  $\sigma \models D$  (see [7]).

*Decidability and Model Checking* Although, validity of full *IDL* is undecidable [8], the validity of *QDDC* formulae is decidable. A tool, called DCVALID, based on an automata-theoretic decision procedure for *QDDC* has been implemented, and found to be effective on many significant examples [7]. In the rest of the paper, we consider a reduction of *IDL* model/validity checking problem to *QDDC* model/validity checking problem. This provides a novel and, in our opinion, a practically useful technique for reasoning about *IDL* properties.

### 3 Digitization

In this section we provide a brief overview of the pioneering work of Henzinger, Manna and Pnueli [4] characterizing the set of systems and properties for which the real-time verification problem is equivalent to integer-time verification.

*Notation* For real numbers  $a, b$  with  $a \leq b$ , let  $[a : b)$  denote the left closed right open interval. Similarly  $(a : b)$ ,  $(a : b]$  and  $[a : b]$ . Let  $\text{frac}(a) \stackrel{\text{def}}{=} a - \lfloor a \rfloor$  denote the fractional part of a real number  $a$ .

**Definition 2 (Digitization).** Let  $x \in \mathbb{R}^0$  and  $\theta = (\sigma, \tau) \in TSS_R$ . Let  $\epsilon \in [0 : 1)$ . Then,  $\epsilon$ -digitization of  $\theta$ , denoted by  $[\theta]_\epsilon$ , is defined as follow.

- $x \downarrow \epsilon \stackrel{\text{def}}{=} \begin{cases} \lfloor x \rfloor & \text{if } \text{frac}(x) \leq \epsilon \\ \lceil x \rceil & \text{else} \end{cases}$
- $[\theta]_\epsilon \stackrel{\text{def}}{=} (\sigma, \tau') \text{ s.t. } \tau'(i) = \tau(i) \downarrow \epsilon$

*Example 2.* Let  $\theta = (\sigma_0, 0.0) \longrightarrow (\sigma_1, 1.5) \longrightarrow (\sigma_2, 4.35) \longrightarrow (\sigma_3, 5.0)$ . Then,  $[\theta]_{0.0} = (\sigma_0, 0) \longrightarrow (\sigma_1, 2) \longrightarrow (\sigma_2, 5) \longrightarrow (\sigma_3, 5)$  and  $[\theta]_{0.4} = (\sigma_0, 0) \longrightarrow (\sigma_1, 2) \longrightarrow (\sigma_2, 4) \longrightarrow (\sigma_3, 5)$ .

**Definition 3.** Let  $\theta \in TSS_R$  and  $\Pi \subseteq TSS_R$ . Then,

- $[\theta] \stackrel{\text{def}}{=} \{[\theta]_\epsilon \mid \epsilon \in [0, 1)\}$
- $[\Pi] \stackrel{\text{def}}{=} \{[\theta]_\epsilon \mid \epsilon \in [0 : 1), \theta \in \Pi\}$ . Note that  $[\Pi] \subseteq TSS_Z$ .
- $Z(\Pi) \stackrel{\text{def}}{=} \Pi \cap TSS_Z$ , the set of integer valued traces of  $\Pi$ .

Set  $[\Pi]$  gives the set of digitized approximations of the behaviours in  $\Pi$  where as  $Z(\Pi)$  gives the integer time fragment of  $\Pi$ . Closures under digitization and inverse digitization, defined below, are used to give a digitization theorem which reduces dense-time model checking to discrete time model checking.

**Definition 4 (Closure under digitization (CD)).**

$CD(\Pi) \stackrel{\text{def}}{=} \forall \theta \in TSS_R. (\theta \in \Pi \Rightarrow (\forall \epsilon \in [0, 1). [\theta]_\epsilon \in \Pi))$ .

**Proposition 1.**  $CD(\Pi)$  iff  $[\Pi] \subseteq \Pi$  iff  $Z(\Pi) = [\Pi]$ .

**Definition 5 (Closure under inverse digitization (CID)).**

$CID(\Pi) \stackrel{\text{def}}{=} \forall \theta \in TSS_R. ((\forall \epsilon \in [0, 1). [\theta]_\epsilon \in \Pi) \Rightarrow \theta \in \Pi)$

**Theorem 1 (Digitization).** Let  $\Psi, \Pi \subseteq TSS_R$ .

- If  $CD(\Psi)$  and  $CID(\Pi)$ , then  $\Psi \subseteq \Pi$  iff  $Z(\Psi) \subseteq Z(\Pi)$
- If  $CID(\Pi)$  then  $\Pi = TSS_R$  iff  $Z(\Pi) = TSS_Z$

Typically, the set  $\Psi$  in the above theorem denotes the set of behaviours of a system where as  $\Pi$  denotes the set of behaviours satisfying some desired property. Hence, the theorem gives sufficient conditions under which the real time verification problem  $\Psi \subseteq \Pi$  can be reduced to the integer time verification problem  $Z(\Psi) \subseteq Z(\Pi)$ . The key requirement for reducing dense-time validity to discrete-time validity is that properties should be closed under inverse digitization CID.

## 4 Digitization of Interval Duration Logic Formulae

We consider the real-time properties specified in logic IDL. We must determine the subset of IDL properties that are closed under inverse digitization (CID).

### 4.1 Closure Properties in IDL

*Notation* Let  $[[D]]_R \stackrel{\text{def}}{=} \{\theta \mid \theta \models D\}$  denote the set of timed state sequences satisfying the IDL formula  $D$ , and let  $[[D]]_Z \stackrel{\text{def}}{=} [[D]]_R \cap TSS_Z$  denote the set of integer timed sequences satisfying  $D$ . Define  $\models_R D \stackrel{\text{def}}{=} [[D]]_R = TSS_R$  and  $\models_Z D \stackrel{\text{def}}{=} [[D]]_Z = TSS_Z$ . Then,  $CID([D]]_R)$  states that the set of timed state sequences satisfying  $D$  is closed under inverse digitization. We shall abbreviate  $CID([D]]_R)$  by  $CID(D)$ . Similarly, also  $CD(D)$  and  $SCID(D)$  (defined below).

**Proposition 2.** *Algebraic properties of CD:*

- (a)  $CD(D_1) \wedge CD(D_2) \Rightarrow CD(D_1 \wedge D_2)$ , (b)  $CD(D_1) \wedge CD(D_2) \Rightarrow CD(D_1 \vee D_2)$ ,
- (c)  $CD(D_1) \wedge CD(D_2) \Rightarrow CD(D_1 \frown D_2)$ , (d)  $CD(D) \Rightarrow CD(\Box D)$ , and
- (e)  $CD(D) \Rightarrow CD(\Diamond D)$ .

**Proposition 3.** *Algebraic properties of CID:*

- (a)  $CID(D_1) \wedge CID(D_2) \Rightarrow CID(D_1 \wedge D_2)$ , and (b)  $CID(D) \Rightarrow CID(\Box D)$ .

Unfortunately, operators  $\vee$ ,  $\frown$ ,  $\neg$  do not preserve the crucial CID property making it difficult to establish that a formula is CID. Below we introduce a stronger notion of closure, SCID, which has vastly superior preservation properties. Also,  $SCID(D)$  implies  $CID(D)$ .

**Definition 6 (Strong Closure under Inverse Digitization(SCID)).** For  $\Pi \subseteq TSS_R$ , let  $SCID(\Pi) \stackrel{\text{def}}{=} \forall \theta \in TSS_R ((\exists \epsilon \in [0, 1). [\theta]_\epsilon \in \Pi) \Rightarrow \theta \in \Pi)$ .

**Proposition 4.** (a)  $SCID(D) \Leftrightarrow CD(\neg D)$ , and (b)  $SCID(D) \Rightarrow CID(D)$ .

**Proposition 5.** *Algebraic properties of SCID:*

- (a)  $SCID(D_1) \wedge SCID(D_2) \Rightarrow SCID(D_1 \wedge D_2)$ ,
- (b)  $SCID(D_1) \wedge SCID(D_2) \Rightarrow SCID(D_1 \vee D_2)$ ,
- (c)  $SCID(D_1) \wedge SCID(D_2) \Rightarrow SCID(D_1 \frown D_2)$ ,
- (d)  $SCID(D) \Rightarrow SCID(\Box D)$  and (e)  $SCID(D) \Rightarrow SCID(\Diamond D)$ . □

**Proposition 6.**  $SCID(D_1) \wedge CID(D_2) \Rightarrow CID(D_1 \vee D_2)$  □

**Lemma 1.** *Formulae of IDL which are free of dense measurements (i.e. QDDC formulae) are CD as well as SCID; hence CID.*

*Proof.* Let  $\theta = (\sigma, \tau)$  and let  $D \in QDDC$ . Then,  $[\theta]_\epsilon = (\sigma, \tau')$ . Note that the interpretation of  $D$  does not depend upon the time stamp sequence  $\tau$ . Hence,  $(\sigma, \tau), [b, e] \models D$  **iff**  $(\sigma, \tau'), [b, e] \models D$ . □

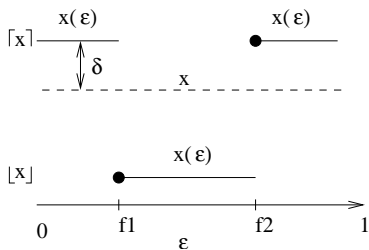


## 4.2 Digitization of Dense Measurements

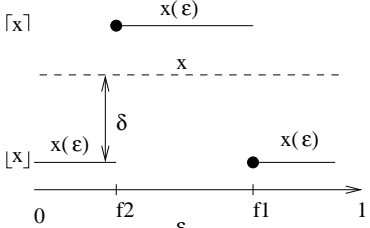
We consider the effect of digitization on dense measurements  $\ell$  and  $\int P$ . We first study some number theoretic properties of digitization.

**Lemma 2.** Let  $c_1 \geq c_2$ . Let  $f_1 = \text{frac}(c_1)$  and  $f_2 = \text{frac}(c_2)$  be the fractional parts of  $c_1$  and  $c_2$ . Let  $\delta = f_1 - f_2$ . Let  $x = c_1 - c_2$  and  $x(\epsilon) = c_1 \downarrow \epsilon - c_2 \downarrow \epsilon$ . We characterize the difference  $x(\epsilon) - x$  for  $0 \leq \epsilon < 1$  below, and plot it alongside.

Let  $f_1 \leq f_2$ . Hence  $\delta \in (-1 : 0]$ . Then,

$$\begin{aligned} \forall \epsilon \in [0 : f_1) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lceil c_1 \rceil, c_2 \downarrow \epsilon = \lceil c_2 \rceil \\ x(\epsilon) - x = -\delta \end{array} \right\} \\ \forall \epsilon \in [f_1 : f_2) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lfloor c_1 \rfloor, c_2 \downarrow \epsilon = \lceil c_2 \rceil \\ x(\epsilon) - x = -(\delta + 1) \end{array} \right\} \\ \forall \epsilon \in [f_2 : 1) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lfloor c_1 \rfloor, c_2 \downarrow \epsilon = \lfloor c_2 \rfloor \\ x(\epsilon) - x = -\delta \end{array} \right\} \end{aligned}$$


Let  $f_1 > f_2$ . Hence  $\delta \in (0 : 1)$ . Then,

$$\begin{aligned} \forall \epsilon \in [0 : f_2) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lceil c_1 \rceil, c_2 \downarrow \epsilon = \lceil c_2 \rceil \\ x(\epsilon) - x = -\delta \end{array} \right\} \\ \forall \epsilon \in [f_2 : f_1) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lceil c_1 \rceil, c_2 \downarrow \epsilon = \lfloor c_2 \rfloor \\ x(\epsilon) - x = -(\delta - 1) \end{array} \right\} \\ \forall \epsilon \in [f_1 : 1) \quad & \left\{ \begin{array}{l} c_1 \downarrow \epsilon = \lfloor c_1 \rfloor, c_2 \downarrow \epsilon = \lfloor c_2 \rfloor \\ x(\epsilon) - x = -\delta \end{array} \right\} \end{aligned}$$


From this it also follows that,

$$\int_0^1 (x(\epsilon) - x) . d\epsilon = 0.0 \quad (1)$$

As a consequence of above case analysis, we have the following three results.

**Proposition 7.** Let  $c_1 \geq c_2$  be non-negative reals, and  $c$  be non-negative integer. Then,

- (A)  $(c_1 - c_2) > c \Rightarrow \exists \epsilon. (c_1 \downarrow \epsilon - c_2 \downarrow \epsilon) > c$
- (B)  $(c_1 - c_2) \geq c \Rightarrow \forall \epsilon. (c_1 \downarrow \epsilon - c_2 \downarrow \epsilon) \geq c$
- (C)  $(c_1 - c_2) \leq c \Rightarrow \forall \epsilon. (c_1 \downarrow \epsilon - c_2 \downarrow \epsilon) \leq c$
- (D)  $(c_1 - c_2) < c \Rightarrow \exists \epsilon. (c_1 \downarrow \epsilon - c_2 \downarrow \epsilon) < c$

*Proof.* The result can be easily seen by examination of Figures in Lemma 2. We omit a detailed algebraic proof.  $\square$

**Theorem 2.**  $CD(\ell \geq c)$  and  $CD(\ell \leq c)$ . Also  $SCID(\ell > c)$  and  $SCID(\ell < c)$ .

*Proof.* We prove that  $CD(\ell \geq c)$ . Proofs of the other parts are omitted. Let  $\theta, [b, e] \models \ell \geq c$ . This implies  $t_e - t_b \geq c$ .

By Proposition 7(B),  $\forall \epsilon \in [0 : 1)$ ,  $(t_e \downarrow \epsilon - t_b \downarrow \epsilon) \geq c$ .

Hence,  $\forall \epsilon \in [0 : 1)$ ,  $[\theta]_\epsilon, [b, e] \models \ell \geq c$ .  $\square$

**Theorem 3.**  $CID(\int P \text{ op } c)$  where  $op \in \{<, \leq, \geq, >\}$ .

*Proof.* By the semantics of  $\int P$ , we have

$$eval(\int P)([\theta]_\epsilon, [b, e]) = \sum_{i=b}^{e-1} \begin{pmatrix} t_{i+1} \downarrow \epsilon - t_i \downarrow \epsilon & \text{if } \sigma, i \models P \\ 0 & \text{otherwise} \end{pmatrix}$$

By Equation 1, we have,

$$\int_0^1 ((t_{i+1} \downarrow \epsilon - t_i \downarrow \epsilon) - (t_{i+1} - t_i)).d\epsilon = 0.0.$$

$$\text{Hence, } \left( \int_0^1 (eval(\int P)([\theta]_\epsilon, [b, e])) - eval(\int P)(\theta, [b, e]) \right).d\epsilon = 0.0$$

Therefore, one of the following must hold

$$\begin{aligned} & - \{ \forall \epsilon. eval(\int P)([\theta]_\epsilon, [b, e]) = eval(\int P)(\theta, [b, e]) \}, \text{ or} \\ & - \{ (\exists \epsilon. eval(\int P)([\theta]_\epsilon, [b, e]) > eval(\int P)(\theta, [b, e])) \wedge \\ & \quad (\exists \epsilon. eval(\int P)([\theta]_\epsilon, [b, e]) < eval(\int P)(\theta, [b, e])) \} \end{aligned}$$

Hence, for  $op \in \{<, \leq, \geq, >\}$  we have,

$$\{ \theta, [b, e] \} \not\models \int P \text{ op } c \Rightarrow \exists \epsilon. [\theta]_\epsilon, [b, e] \not\models \int P \text{ op } c \}.$$

The result follows immediately from this.  $\square$

### 4.3 Proving IDL Formulae CID

The algebraic properties of the previous section can be used to infer that an IDL formula is *SCID* or *CID* purely from its syntactic structure. Recall that  $IDL_\ell$  is the IDL subset without  $\int P$  terms. Formulae of form  $(\ell \text{ op } c)$  will be called *length constraints*.

*Example 3.*  $CID(\Box(\ell \leq 60 \Rightarrow \int Leak \leq 5))$ .

*Proof.* By Theorem 2,  $CD(\ell \leq 60)$ . Hence, by Proposition 4(a),  $SCID(\neg(\ell \leq 60))$ . Also, by Theorem 3,  $CID(\int Leak \leq 5)$ . Hence using Propositions 6 and 3(b), we have  $CID(\Box(\neg(\ell \leq 60) \vee \int Leak \leq 5))$  which gives the result.  $\square$

**Theorem 4.** For  $D \in IDL_l$ , if the following conditions hold then  $SCID(D)$ .

- every length constraint occurring within the scope of even number negation has the form  $\ell > c$  or  $\ell < c$ , and
- every length constraint occurring within the scope of odd number of negations has the form  $\ell \leq c$  or  $\ell \geq c$ .  $\square$

Using the Digitization Theorem 1, we get the following Theorem.

**Theorem 5.** If  $CID(D)$  then  $\models_R D$  iff  $\models_Z D$ .

*Digitization Approximation of  $IDL_l$  formulae* Not all  $IDL_l$  formulae are SCID. We now define strengthening and weakening transformations ST and WT of  $IDL_l$  formulae which result in SCID formulae.

**Definition 7.**  $ST(D) \stackrel{\text{def}}{=} \text{Substituting in } D \text{ every atomic '}\ell\text{' constraint}$   
 $(\ell \geq c), \text{ under an even number of negations, to } (\ell > c)$   
 $(\ell \leq c), \text{ under an even number of negations, to } (\ell < c)$   
 $(\ell > c), \text{ under an odd number of negations, to } (\ell \geq c)$   
 $(\ell < c), \text{ under an odd number of negations, to } (\ell \leq c)$

**Definition 8.**  $WT(D) \stackrel{\text{def}}{=} \text{Substituting in } D \text{ every atomic '}\ell\text{' constraint}$   
 $(\ell \geq c) \text{ under an even number of negations to } (\ell > c - 1)$   
 $(\ell \leq c) \text{ under an even number of negations to } (\ell < c + 1)$   
 $(\ell > c) \text{ under an odd number of negations to } (\ell \geq c + 1)$   
 $(\ell < c) \text{ under an odd number of negations to } (\ell \leq c - 1)$

*Example 4.* Refer to the formulae of Example 1. Using Theorem 4, we can conclude that  $SCID(\text{Follows}(P, Q, d))$  and  $CD(\text{FollowsWk}(P, Q, d))$ . Using the definitions of  $WT$  and  $ST$  we get that

$$ST(\text{FollowsWk}(P, Q, d)) = \text{Follows}(P, Q, d), \text{ and} \\ WT(\text{FollowsWk}(P, Q, d)) = \text{Follows}(P, Q, d + 1). \quad \square$$

**Theorem 6.** For every  $D \in IDL_l$ , we have

1.  $\models_R ST(D) \Rightarrow D$  and  $\models_R D \Rightarrow WT(D)$ .
2.  $SCID(ST(D))$  and  $SCID(WT(D))$ .  $\square$

By combining the above with Theorem 5 we obtain a method of reducing IDL validity to ZIDL validity. This is outlined in the following theorem, which also suggests how to promote counter examples from discrete-time to dense-time. Validity of *ZIDL* is decidable as shown in the next section.

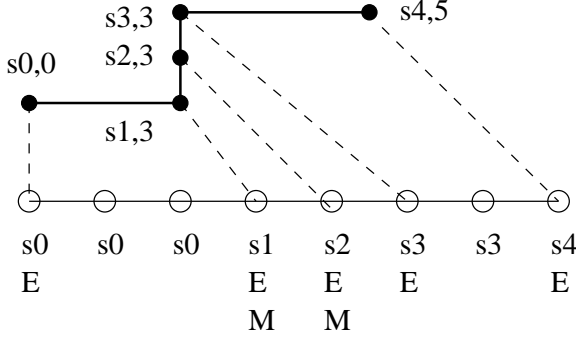
**Theorem 7.** For any formula  $D \in IDL_l$ ,

1.  $\models_Z ST(D) \Rightarrow \models_R D$
2.  $\theta \not\models_Z WT(D) \Rightarrow \theta \not\models_R D$ .
3.  $\models_Z WT(D) \Rightarrow \models_R WT(D)$ .

## 5 ZIDL to QDDC

We now consider a reduction from ZIDL to QDDC with the aim that we can utilize tools for QDDC to reason about ZIDL. Note that ZIDL is a logic of weakly monotonic integer-timed state sequences where as QDDC is a logic of untimed state sequences.

We first define an encoding  $\alpha$  of a  $TSS_Z$  behaviour by means of a untimed sequence of states. This is depicted in Figure 5. The line with dark circles represents  $TSS_Z$  behaviour  $\theta = (s_0, 0) \rightarrow (s_1, 3) \rightarrow (s_2, 3) \rightarrow (s_3, 3) \rightarrow (s_4, 5)$ . The bottom line denotes QDDC behaviour  $\alpha(\theta)$ . A function *place* maps positions in  $\theta$  to corresponding positions in  $\alpha(\theta)$  and is denoted by dashed lines. A new boolean variable  $E$  marks exactly the positions of  $\alpha(\theta)$  which are image of *place*. Note that  $\theta$  is weakly monotonic having “micro steps” which do not change the time stamp. A new boolean variable  $M$  marks exactly the positions in  $\alpha(\theta)$  where the next step is a micro step. If  $Pvar$  is the set of original



**Fig. 1.** Encoding of ZIDL Behaviours

propositional variables, then  $ST_{E,M} = 2^{(Pvar \cup \{E,M\})}$  denotes the states assigning truth values to  $Pvar \cup \{E, M\}$ . We omit the formal definition of encoding  $\alpha : TSS_Z \rightarrow (ST_{E,M})^+$  which can be found in the full paper.

Not all elements of  $ST_{E,M}^+$  correspond to  $TSS_Z$  behaviours. We give a formula **CONSIST** specifying consistent behaviours. Every consistent QDDC behaviour *uniquely* denotes a ZIDL behaviour and vice versa.

$$\begin{aligned} CONSIST(Pvar) &\stackrel{\text{def}}{=} \\ &\{ ([E]^0 \wedge true \wedge [E \wedge \neg M]^0) \wedge (\Box \llbracket M \Rightarrow E \rrbracket) \wedge \\ &(\neg \Diamond ([M]^0 \wedge (\eta = 1) \wedge [\neg E]^0)) \wedge \\ &\forall p \in Pvar. (\Box ([E]^0 \wedge (\eta = 1) \wedge \llbracket \neg E \rrbracket \Rightarrow (\llbracket p \rrbracket \vee \llbracket \neg p \rrbracket))) \} \end{aligned}$$

**Proposition 8.** (1)  $\forall \theta \in TSS_Z. \alpha(\theta) \models CONSIST$ .

(2) Map  $\alpha : TSS_Z \rightarrow \{\sigma \in ST_{E,M}^+ \mid \sigma \models CONSIST\}$  is an isomorphism.  $\square$

We now give the translation  $(\beta)$  of ZIDL formulae into QDDC formulae over  $ST_{E,M}^+$ , where  $\beta$  is overloaded to map ZIDL measurement terms to QDDC expressions also.

**Definition 9.**

$$\begin{aligned} \beta(\eta) &= \sum E & \beta(\sum P) &= \sum (E \wedge P) \\ \beta(\ell) &= \sum (\neg M) & \beta(\int P) &= \sum (\neg M \wedge P) \end{aligned}$$

**Proposition 9.**  $eval(t)(\theta, [b, e]) = eval(\beta(t))(\alpha(\theta), [place(b), place(e)])$   $\square$

Since  $\alpha(\theta)$  has positions (intervals) which do not correspond to positions (intervals) of  $\theta$ , we translate the formulae ensuring that all chopping points correspond to pre-existing positions in  $\theta$ , i.e. points where  $E$  is true.

**Definition 10.**

$$\begin{aligned} \beta([P]^0) &= [P]^0 & \beta([P]) &= [P] \\ \beta(t \text{ op } c) &= \beta(t) \text{ op } c & \beta(\neg D) &= \neg \beta(D) \\ \beta(D_1 \cap D_2) &= \beta(D_1) \cap [E]^0 \cap \beta(D_2) & \beta(D_1 \wedge D_2) &= \beta(D_1) \wedge \beta(D_2) \end{aligned}$$

**Theorem 8.** Let  $\theta \in TSS_Z$ . Then,

$$\theta, [b, e] \models_Z D \quad \text{iff} \quad \alpha(\theta), [place(b), place(e)] \models_{QDDC} \beta(D) \quad \square$$

**Theorem 9.** (1)  $\models_Z D \quad \text{iff} \quad \models_{QDDC} (CONSIST(Pvar) \Rightarrow \beta(D))$

$$(2) \sigma \not\models_{QDDC} (CONSIST \Rightarrow \beta(D)) \text{ then } \alpha^{-1}(\sigma) \not\models_Z D. \quad \square$$

Since, validity of  $QDDC$  is decidable [7], we have the following corollary.

**Corollary 1.** Validity of  $ZIDL$  formulae is decidable.  $\square$

## 6 Verification by Digitization: An Example

It is our belief that techniques developed in this paper are of practical importance. These techniques allow dense-time properties to be checked by reducing them to discrete-time properties which can be efficiently analysed. We illustrate this approach by proving validity of a small IDL formula.

Recall the formulae  $Follows(P, Q, d)$  and  $FollowsWk(P, Q, d)$  given in Examples 1 and 4. Let,  $Within(P, Q, d) \stackrel{\text{def}}{=} \Box((\llbracket P \rrbracket \wedge (\ell \geq d)) \Rightarrow \Diamond \llbracket Q \rrbracket^0)$ . It states that in any interval with  $P$  invariantly true and having time length of  $d$  or more, there must be some position with  $Q$  true. Our aim is to check the validity of the following  $GOAL$  formula for various integer values of  $d_1, d_2$ .

$$GOAL \stackrel{\text{def}}{=} Follows(P, Q, d_1) \Rightarrow Within(P, Q, d_2)$$

Unfortunately,  $SCID(GOAL)$  does not hold and we cannot reduce the problem to equivalent  $QDDC$  validity checking. However, we can use the digitization approximation technique. We compute  $QDDC$  approximations  $\beta(ST(GOAL))$  and  $\beta(WT(GOAL))$  of the IDL formula  $GOAL$  using Definitions 7, 8, 9, 10.

$$\begin{aligned} \beta(ST(GOAL)) &= \beta(FollowsWk(P, Q, d_1)) \Rightarrow \beta(Within(P, Q, d_2)) \\ \beta(WT(GOAL)) &= \beta(FollowsWk(P, Q, d_1 - 1)) \Rightarrow \beta(Within(P, Q, d_2)) \\ \beta(FollowsWk(P, Q, d)) &= \neg(true \frown [E]^0 \frown \\ &\quad ((\llbracket P \rrbracket \wedge (\Sigma \neg M > d)) \frown [E]^0 \frown [\neg Q]^0) \frown [E]^0 \frown true) \\ \beta(Within(P, Q, d)) &= \neg(true \frown [E]^0 \frown \\ &\quad ((\llbracket P \rrbracket \wedge (\Sigma \neg M \geq d)) \wedge \neg(true \frown [Q \wedge E]^0 \frown true)) \frown [E]^0 \frown true) \end{aligned}$$

The resulting formulae can be analysed using the  $QDDC$  validity checker  $DCVALID$  [7] for various constants  $d_1, d_2$ .

*Experimental Verification with DCVALID* The verification was carried out using  $DCVALID1.4$  tool running on Pentium4 1.4GHz PC system running Linux 2.4.18 kernel.

*Case 1* For  $d_1 = 10, d_2 = 12$ , the validity checker returned the result that  $\models_{QDDC} CONSIST(P, Q) \Rightarrow \beta(ST(GOAL))$ .

Its verification took 1.57 seconds of CPU time. From this, by using Theorems 9(1) and 7(1), we concluded that  $\models_R GOAL$ .

*Case 2* For  $d_1 = 10$ ,  $d_2 = 7$ , the validity checker returned the result that  $\not\models_{QDDC} CONSIST(P, Q) \Rightarrow \beta(ST(GOAL))$ . The tool gave a counter example, but as this is not guaranteed to be a counter example for the original formula  $GOAL$ , we disregarded it. Instead, we invoked the tool with the weak approximation  $\beta(WT(GOAL))$ . The validity checker returned the result that  $\not\models_{QDDC} CONSIST \Rightarrow \beta(WT(GOAL))$  and gave the following counter example in 0.17 seconds of CPU time.

MT	00000000
ES	10000001
P	11111110
Q	00000000

This corresponds to the IDL behaviour  $\theta = (P \wedge \neg Q, 0) \rightarrow (\neg P \wedge \neg Q, 7)$ . By using Theorems 9(2) and 7(3) we concluded that  $\theta \not\models_R GOAL$ . Thus, we generated a counter-example for  $GOAL$ .

One limitation of our method is that it would fail to conclude anything about  $\models_R GOAL$  in case we get  $\not\models_Z \beta(ST(GOAL))$  and  $\models_Z \beta(WT(GOAL))$ , as this would only establish that  $\not\models_R ST(D)$  and  $\models_R WT(D)$ .

## 7 Related Work

The technique of digitization is aimed at reducing a dense-time verification problem to a discrete-time verification problem. There is some experimental evidence that, with existing techniques, automatic verification of discrete-time systems may be significantly easier than the verification of similar dense-time systems [2]. This makes digitization a practically important approach for the verification of dense-time properties.

In their pioneering work Henzinger, Manna and Pnueli [4] proposed the digitization technique for the verification of dense-time properties, and formulated the digitization theorem which gives sufficient conditions for reducing the model checking of dense-time properties to the model checking of discrete-time properties. In particular, they showed that for CID properties, dense-time validity and discrete-time validity are equivalent. They also studied some properties of logic MTL which are CID.

In this paper, we have considered digitization of IDL properties. It is quite hard to establish which IDL formulae are CID. To obviate this, we have given a new notion of *Strong Closure Under Inverse Digitization* (SCID) which implies CID. Almost all operators of IDL preserve SCID, giving us powerful structural characterisations of formulae which are SCID (see Theorem 4). Moreover, for formulae which are not SCID, we have given approximations to stronger and weaker formulae which are SCID (Theorem 7). IDL is a highly expressive dense-time logic with a powerful  $\frown$  (chop) operator and a notion of  $\int P$  giving the accumulated amount of time for which proposition  $P$  holds in a time interval. Digitization (CID) of such properties is one of the main contributions of this paper. For example, we have shown that the “critical duration” formulae [9]

like  $\Box(\ell \leq c \Rightarrow \int P \leq d)$  are CID. Such formulae are quite hard to verify in dense-time.

Digitization reduces verification of IDL properties to verification of ZIDL properties. ZIDL formulae are interpreted over weakly monotonic integer-timed state sequences. In our next reduction, we have translated ZIDL formulae to “equivalent” QDDC formulae (Theorem 9). The translation preserves models under a suitable isomorphism. This also establishes the decidability of logic ZIDL. The use of the “count” construct  $\Sigma P$  of QDDC in this translation is significant.

Putting all these together, we are able to mechanically reduce the validity of IDL to the validity of QDDC for a large class formulae, and to approximate this reduction in other cases. We have illustrated the use of this technique by a small example in Section 6. A more extensive example of the verification of a Minepump Specification [8] can be found in the full version of this paper.

Digitization of Duration calculus has been studied before. Under bounded-variability assumption Franzle [3] has shown the decidability of Duration Calculus. Hung and co-authors have modelled digitized behaviours directly within DC and used axioms of DC to reason about digitized behaviours [5].

## References

1. R. Alur and D. L. Dill: Automata for modeling Real-time systems. In: 17<sup>th</sup> ICALP, Lecture Notes in Computer Science, Vol 443. Springer-Verlag (1990)
2. D. Beyer: Rabiit: Verification of Real-Time Systems. In: *Workshop on Real-time Tools (RT-TOOLS'2001)*. Aalborg, Denmark (2001)
3. M. Franzle: Decidability of Duration Calculi on Restricted Model Classes. *ProCoS Technical Report Kiel MF/1*. Christian-Albrechts Universität Kiel, Germany (1996)
4. T. A. Henzinger, Z. Manna and A. Pnueli: What good are digital clocks?. In: *ICALP'92*, Lecture Notes in Computer Science, Vol 623. Springer-Verlag (1992) 545-558
5. D. V. Hung and P. H. Giang: Sampling Semantics of Duration Calculus. In: *FTRTFT'96*, Lecture Notes in Computer Science, Vol 1135. Springer-Verlag (1996) 188-207
6. P.K. Pandya and D.V. Hung: A Duration Calculus of Weakly Monotonic Time, In: A.P.Ravn and H. Rischel (eds.): *FTRTFT'98*. Lecture Notes in Computer Science, Vol 1486. Springer-Verlag (1998)
7. P.K. Pandya: Specifying and Deciding Quantified Discrete-time Duration Calculus Formulae using DCVALID: An Automata Theoretic Approach. In: *Workshop on Real-time Tools (RTTOOLS'2001)*. Aalborg, Denmark (2001)
8. P.K. Pandya: Interval Duration Logic: Expressiveness and Decidability. In: *Workshop on Theory and Practice of Timed Systems (TPTS'2002)*, Electronic Notes in Theoretical Computer Science, ENTCS **65.6**. Elsevier Science B.V. (2002)
9. A.P. Ravn: Design of Real-time Embedded Computing Systems. Department of Computer Science, Technical University of Denmark (1994)
10. Zhou Chaochen, C. A. R. Hoare and A. P. Ravn. A Calculus of Durations, *Information Processing Letters*, **40**(5). (1991) 269-276

# Timed Control with Partial Observability<sup>\*</sup>

Patricia Bouyer<sup>1\*\*</sup>, Deepak D’Souza<sup>2\*\*\*</sup>, P. Madhusudan<sup>3†</sup>, and Antoine Petit<sup>1</sup>

<sup>1</sup> LSV – CNRS UMR 8643 & ENS de Cachan, 61, av. du Prés. Wilson, 94230 Cachan, France

<sup>2</sup> Chennai Mathematical Institute, 92 G.N. Chetty Road, Chennai 600 017, India

<sup>3</sup> University of Pennsylvania, C.I.S. Dept., Philadelphia, USA

bouyer@lsv.ens-cachan.fr, deepak@cmi.ac.in,

madhusudan@sacl.cis.upenn.edu, petit@lsv.ens-cachan.fr

**Abstract.** We consider the problem of synthesizing controllers for timed systems modeled using timed automata. The point of departure from earlier work is that we consider controllers that have only a partial observation of the system that it controls. In discrete event systems (where continuous time is not modeled), it is well known how to handle partial observability, and decidability issues do not differ from the complete information setting. We show however that timed control under partial observability is undecidable even for internal specifications (while the analogous problem under complete observability is decidable) and we identify a decidable subclass.

## 1 Introduction

In the last twenty-five years, system verification has become a very active field of research in computer science, with numerous success stories. A natural generalization of system verification is the *control of systems*, which is useful in the context of automated system design. The problem here is not to verify that the system meets a given specification, but to *control* the system in such a way that the specification is met. In this framework a system, often called a plant, is usually viewed as *open* and interacting with a “hostile” environment. The problem then is to come up with a controller such that no matter how the environment behaves, the controlled plant satisfies the given specification. An important issue concerns the power of the controller, both in terms of controllability and observability. The controller can act only on a subset of the actions of the plant, referred to as the controllable actions. Depending on the nature of the plant, the non-controllable actions (also called environment actions) could all be observable (*full observability*) or only a proper subset (*partial observability*) may be observable by the controller.

The computer science community has studied these problems in the setting of automated synthesis, which can be viewed as a special case of control synthesis. In the case of untimed (*i.e.* discrete) systems, this problem is now well understood, both for

---

<sup>\*</sup> With the partial support of the French-Indian project CEFIPRA n°2102–1

<sup>\*\*</sup> This work was partly carried out while author had a post-doctoral position at BRICS, Aalborg University (Denmark).

<sup>\*\*\*</sup> Part of this work was done during a visit to LSV, ENS de Cachan (France).

<sup>†</sup> This research was supported by NSF award CCR99-70925 and NSF award ITR/SY 0121431.



full observability (in terms of two-player games of complete information) [Tho02], and for partial observability [KG95, Rei84, KV97].

In parallel, there has been a growing importance of verification for real-time systems, and this leads to the natural question of whether techniques developed in the untimed setting for the controller synthesis problem can be generalized to timed systems (see for example the papers [AMPS98, WTH91, DM02, FLM02]). In this framework, the timed system is usually given by a timed transition system (*i.e.* a timed automaton as defined by Alur and Dill [AD94] but without acceptance conditions). As in the untimed case, various proposals have been made and studied for the specification. For instance, in [AMPS98, WTH91] the specification is given as an internal winning condition on the state-space of the plant. External specifications given by timed automata [DM02] or by TCTL-formulas [FLM02] have also been investigated. Note that since we deal here with classes of specifications which are not, in general, closed under complementation, a distinction has to be made between specifications that describe desired behaviours and those that define undesired behaviours. The decidability of the problem can depend crucially on this choice [DM02].

An important and new issue in the timed framework is related to the resources allowed to the controller. By a controller's resources we mean a description of the number of new clocks available to the controller and the granularity or fineness of the guards it can use. As done in [AMPS98, WTH91, FLM02], a simple and possible assumption is that these resources are not fixed: the controller can use arbitrary new clocks and guards that compare the clocks to any constant. Another probably more realistic setting is that the resources of the controller are fixed *a priori*, as proposed in [DM02]. The important point is that the controller synthesis problem becomes simpler in the latter case [DM02, CHR02].

The purpose of this paper is to investigate the problem of timed controller synthesis under the constraint of partial observability. In the timed setting, the partial observability assumption applies not only to uncontrollable actions but also to the clocks of the system. The setting of [AMPS98] and [DM02] treat the full observability hypothesis and their main results are summarized in the table below.

Full observability hypothesis			
Resources	Det. Spec. (Internal/External)	External Non-deterministic Spec.	
		Desired behaviours	Undesired behaviours
Fixed	Decidable [WTH91, AMPS98]	Undecidable [DM02]	Decidable [DM02]
Non-fixed	Decidable [DM02]	Undecidable [DM02]	Undecidable [DM02]

Of course, when we drop the hypothesis of full observability, the undecidable cases carry over under the weaker assumption of partial observability.

While full and partial observability lead to the same decidable cases in the untimed framework [KV97], our results show that the situation is rather different in the timed setting. Indeed, if the resources of the controller are not fixed *a priori*, the problem becomes undecidable, even for *deterministic* specifications. This is in contrast to the complete observability setting, where the problem is decidable for deterministic specifications [DM02]. However, if the resources of the controller are fixed, the timed controller synthesis problem remains decidable both for deterministic specifications, and

for non-deterministic specifications that specify undesired behaviours. The results in this paper thus complete the study begun in [AMPS98,DM02]. The completed picture is summarized in the next table, with the new results of this paper written in bold face. Note that in this timed setting, [LW95,WT97] have studied the problem though in a different framework from ours. In [LW95] time is modeled discretely via an explicit clock tick, while [WT97] considers internal specifications on the state space of the plant and provides only semi-decision procedures.

Partial observability hypothesis			
Resources	Det. Spec. (Internal/External)	External Non deterministic Spec.	
		Desired behaviours	Undesired behaviours
Fixed	<b>DECIDABLE</b>	Undecidable [DM02]	<b>DECIDABLE</b>
Non-fixed	<b>UNDECIDABLE</b>	Undecidable [DM02]	Undecidable [DM02]

The undecidability results are obtained through a reduction of the universality problem for timed automata [AD94] and are presented in Section 4. The technique used for the main decision procedure can be viewed as a generalization of the technique used in [DM02] and is presented in Section 5.

Due to lack of space proofs are omitted; details can be found in [BDMP02].

## 2 Preliminaries

For a finite alphabet  $\Sigma$ , let  $\Sigma^*$  (resp.  $\Sigma^\omega$ ) be the set of *finite* (resp. *infinite*) sequences of elements in  $\Sigma$ . We use  $\Sigma^\infty$  to denote  $\Sigma^* \cup \Sigma^\omega$ . The length of an element  $\alpha$  of  $\Sigma^\infty$  is denoted  $|\alpha|$  (if  $\alpha \in \Sigma^\omega$ , we set  $|\alpha| = \omega$ ).

**Timed Words.** We consider a finite set of *actions*  $\Sigma$  and as time domain the set  $\mathbb{R}_{\geq 0}$  of non-negative reals. A *timed word*  $\omega = (a_i, t_i)_{1 \leq i}$  is an element of  $(\Sigma \times \mathbb{R}_{\geq 0})^\infty$  which satisfies:

- *Monotonicity*:  $\forall i < j, t_i < t_j$
- If  $\omega$  is infinite, *non-zenoness*:  $\forall t \in \mathbb{R}_{>0}, \exists i, t_i > t$

We denote the set of finite (infinite resp.) timed words over  $\Sigma$  by  $T\Sigma^*$  ( $T\Sigma^\omega$  resp.), and set  $T\Sigma^\infty = T\Sigma^* \cup T\Sigma^\omega$ .

We consider a finite set  $X$  of variables, called *clocks*. A *clock valuation* over  $X$  is a mapping  $v : X \rightarrow \mathbb{R}_{\geq 0}$  that assigns to each clock a time value. We use  $\mathbf{0}$  to denote the zero-valuation which resets each  $x$  in  $X$  to 0. If  $t \in \mathbb{R}_{\geq 0}$ , the valuation  $v + t$  is defined as  $(v + t)(x) = v(x) + t, \forall x \in X$ . If  $Y$  is a subset of  $X$ , the valuation  $v[0/Y]$  is defined as: for each clock  $x$ ,  $(v[0/Y])(x) = 0$  if  $x \in Y$  and is  $v(x)$  otherwise.

The set of *constraints* (or *guards*) over a set of clocks  $X$ , denoted  $\mathcal{G}(X)$ , is given by the syntax

$$g ::= (x \sim c) \mid \neg g \mid (g \vee g) \mid (g \wedge g)$$

where  $x \in X$ ,  $c$  is an element of the set  $\mathbb{Q}_{\geq 0}$  of non-negative rationals and  $\sim$  is one of  $<, \leq, =, \geq$ , or  $>$ . In this paper, we write  $v \models g$  if the valuation  $v$  satisfies the clock constraint  $g$ . The set of valuations over  $X$  which satisfy a guard  $g \in \mathcal{G}(X)$  is denoted by  $\llbracket g \rrbracket_X$ , or just  $\llbracket g \rrbracket$  when the set of clocks is clear from the context.

**Symbolic Alphabet and Timed Automata.** Let  $\Sigma$  be an alphabet of actions, and  $X$  be a finite set of clocks. A *symbolic alphabet*  $\Gamma$  based on  $(\Sigma, X)$  is a finite subset of  $\Sigma \times \mathcal{G}(X) \times 2^X$ . As used in the framework of timed automata [AD94], a symbolic word  $\gamma = (b_i, g_i, Y_i)_{i \geq 1} \in \Gamma^\infty$  gives rise to a set of timed words, denoted  $tw(\gamma)$ . We interpret the symbolic action  $(a, g, Y)$  to mean that action  $a$  can happen if the guard  $g$  is satisfied, with the clocks in  $Y$  being reset after the action. Formally, let  $\sigma = (a_i, t_i)_{i \geq 1} \in T\Sigma^\infty$ . Then  $\sigma \in tw(\gamma)$  if there exists a sequence  $v = (v_i)_{i \geq 1}$  of valuations such that (with the notations  $t_0 = 0$  and  $v_0 = \mathbf{0}$ ):

$$|\sigma| = |\gamma| = |v| \quad \text{and} \quad \forall i \geq 1, \begin{cases} a_i = b_i \\ v_{i-1} + (t_i - t_{i-1}) \models \varphi_i \\ v_i = (v_{i-1} + (t_i - t_{i-1}))[0/Y_i] \end{cases}$$

A timed automaton over  $(\Sigma, X)$  is a tuple  $\mathcal{A} = (\mathcal{T}, F, \mathcal{F})$  where  $\mathcal{T} = (Q, q_0, \rightarrow)$ , with  $\rightarrow \subseteq Q \times \Gamma \times Q$ , is a finite state transition system over some symbolic alphabet  $\Gamma$  based on  $(\Sigma, X)$ ,  $F \subseteq Q$  is the set of final states and  $\mathcal{F}$  is an *acceptance condition* for infinite behaviours. We consider instances of  $\mathcal{F}$  as a *Büchi condition*, specified by a subset  $B$  of repeated states, or a *parity condition*, specified by  $ind : Q \rightarrow \{0, \dots, d\}$  (where  $d \in \mathbb{N}$ ), that assigns an *index* to each state.

The timed automaton  $\mathcal{A}$  (or the transition system  $\mathcal{T}$ ) is said to be *deterministic* if, for every state, the set of symbolic actions enabled at that state is time-deterministic *i.e.* do not contain distinct symbolic actions  $(a, g, Y)$  and  $(a, g', Y')$  with  $\llbracket g \rrbracket \cap \llbracket g' \rrbracket \neq \emptyset$ .

A *path* in  $\mathcal{T}$  is a finite or an infinite sequence of consecutive transitions:

$$P = q_0 \xrightarrow{a_1, g_1, Y_1} q_1 \xrightarrow{a_2, g_2, Y_2} q_2 \dots, \text{ where } (q_{i-1}, a_i, g_i, Y_i, q_i) \in \rightarrow, \forall i > 0$$

The path is said to be *accepting* in  $\mathcal{A}$  if *either* it is finite and it ends in a final state, *or* it is infinite and the set  $inf(P)$ , which consists of the states which appear infinitely often in  $P$ , satisfies:

- $inf(P) \cap B \neq \emptyset$  (in case of a Büchi condition)
- $\min(\{ind(q) \mid q \in inf(P)\})$  is an even number (in case of a parity condition)

A timed automaton  $\mathcal{A}$  can be interpreted as a classical finite automaton<sup>1</sup> on the symbolic alphabet  $\Gamma$ . Viewed as such,  $\mathcal{A}$  accepts (or generates) a language of symbolic words,  $L_{\text{symp}}(\mathcal{A}) \subseteq \Gamma^\infty$ , constituted by the labels of the accepting paths in  $\mathcal{A}$ . But we will be more interested in the timed language generated by  $\mathcal{A}$ , denoted  $L(\mathcal{A})$ , and defined by  $L(\mathcal{A}) = tw(L_{\text{symp}}(\mathcal{A}))$ .

The set of finite symbolic words generated by a timed automaton  $\mathcal{A}$ ,  $L_{\text{symp}}(\mathcal{A}) \cap \Gamma^*$  will be denoted by  $L_{\text{symp}}^*(\mathcal{A})$ . Similarly,  $L^*(\mathcal{A}) = L(\mathcal{A}) \cap T\Sigma^*$ .

Let  $\mathcal{T}$  be a timed transition system on the symbolic alphabet  $\Gamma$ . We define  $L_{\text{symp}}(\mathcal{T})$  as the set  $L_{\text{symp}}(\mathcal{A})$  where  $\mathcal{A}$  is the timed automaton obtained from  $\mathcal{T}$  by setting all states to be both final and repeated. The languages  $L(\mathcal{T})$ ,  $L_{\text{symp}}^*(\mathcal{T})$  and  $L^*(\mathcal{T})$  are defined in a similar way.

<sup>1</sup> We assume the standard definitions of classical finite (untimed) automata on finite and infinite words, or (untimed) transition systems.

**Synchronized Product.** We define the *synchronized product* of timed transition systems. Let  $\Sigma = \bigcup_{i \in \{1, \dots, k\}} \Sigma_i$  be an alphabet. Let  $X = \bigcup_{i \in \{1, \dots, k\}} X_i$  be a finite set of clocks and let  $(R_i)_{1 \leq i \leq k}$  be a partition of  $X$ . And, for  $i = 1, \dots, k$ , let  $\mathcal{T}_i = (Q_i, q_0^i, \longrightarrow_i)$  be a timed transition system on some symbolic alphabet over  $(\Sigma_i, X_i)$  with resets only in  $R_i$ . The *synchronized product* of  $\mathcal{T}_1, \dots, \mathcal{T}_k$  w.r.t. the distributed clock-alphabet  $((\Sigma_1, X_1, R_1), \dots, (\Sigma_k, X_k, R_k))$ , written  $\mathcal{T}_1 \parallel \dots \parallel \mathcal{T}_k$ , is defined to be the transition system  $\mathcal{T} = (Q, q_0, \longrightarrow)$ , where  $Q = Q_1 \times \dots \times Q_k$ ,  $q_0 = (q_0^1, \dots, q_0^k)$  and the transitions are constructed as follows. Let  $(q_1, \dots, q_k)$  be a state of  $Q$ , let  $a \in \Sigma$  and, for every  $i$  such that  $a \in \Sigma_i$ , let  $q_i \xrightarrow{a, g_i, Y_i} q'_i$  be a transition in  $\mathcal{T}_i$ . Then there exists in  $\mathcal{T}$  a (synchronized) transition  $(q_1, \dots, q_k) \xrightarrow{a, g, Y} (\bar{q}_1, \dots, \bar{q}_k)$  with

- $g = \bigwedge_{i|a \in \Sigma_i} g_i$
- $Y = \bigcup_{i|a \in \Sigma_i} Y_i$
- for any  $i = 1, \dots, k$ ,  $\bar{q}_i = q'_i$  if  $a \in \Sigma_i$  and  $\bar{q}_i = q_i$  otherwise.

In the sequel, we will use the notations  $(a, g, Y) \upharpoonright j = \epsilon$  if  $a \notin \Sigma_j$ ,  $(a, g, Y) \upharpoonright j = (a, g_j, Y_j)$  otherwise. We extend this projection to words over symbolic product actions in the obvious way, interpreting  $\epsilon$  as the empty string.

The idea is that on an action in  $\Sigma$ , the agents involved in the action make a synchronized move. Also, each transition system  $\mathcal{T}_i$  can read the clocks  $X_i$  but a clock  $x \in X$  can be reset only by one agent (the agent  $j$  such that  $x \in R_j$ ).

**Granularity and Regions.** We define a measure of the clocks and constants used in a set of constraints, called its *granularity*. A *granularity* is a tuple  $\mu = (X, m, max)$  where  $X$  is a set of clocks,  $m$  is a positive integer and  $max : X \rightarrow \mathbb{Q}^+$  a function which associates with each clock of  $X$  a positive rational number. The granularity of a finite set of constraints is the tuple  $(X, m, max)$  where  $X$  is the exact set of clocks mentioned in the constraints,  $m$  is the least common multiple of the denominators of the constants mentioned in the constraints, and  $max$  records for each  $x \in X$  the largest constant it is compared to. A constraint  $g$  is said  $\mu$ -granular if it belongs to some set of constraints of granularity  $\mu$  (note that a  $\mu$ -granular constraint is also  $\nu$ -granular for any granularity  $\nu$  finer than  $\mu$ ). We denote the set of all  $\mu$ -granular constraints by  $\mathcal{G}(\mu)$ . A constraint  $g \in \mathcal{G}(\mu)$  is said  $\mu$ -atomic if for all  $g' \in \mathcal{G}(\mu)$ , either  $\llbracket g \rrbracket \subseteq \llbracket g' \rrbracket$  or  $\llbracket g \rrbracket \cap \llbracket g' \rrbracket = \emptyset$ . Let  $atoms_\mu$  denote this set of  $\mu$ -atomic constraints.

By the granularity of a timed automaton (or a timed transition system), we will mean the granularity of the set of constraints used in it.

For granularities  $\mu = (X, m, max)$  and  $\nu = (X', m', max')$  we use  $\mu + \nu$  to mean the combined granularity of  $\mu$  and  $\nu$  which is  $(X \cup X', lcm(m, m'), max'')$  where  $max''(x)$  is the larger of  $max(x)$  and  $max'(x)$ , assuming  $max(x) = 0$  for  $x \in X' - X$ , and  $max'(x) = 0$  for  $x \in X - X'$ .

Let  $\mu = (X, m, max)$  be a granularity. A  $\mu$ -region is thus an equivalence class of valuations over  $X$  which satisfy

- for each  $x \in X$ , either both  $v(x), v'(x) > max(x)$ , or  $\lfloor m.v(x) \rfloor = \lfloor m.v'(x) \rfloor$  and  $frac(m.v(x)) = 0$  iff  $frac(m.v'(x)) = 0$ . By  $\lfloor t \rfloor$  we mean the integer part of  $t$  and by  $frac(t)$  we mean the value  $t - \lfloor t \rfloor$ .

- for each pair of clocks  $x, y$  in  $X$  with  $v(x), v'(x) \leq \max(x)$  and  $v(y), v'(y) \leq \max(y)$ ,  $\text{frac}(m.v(x)) = \text{frac}(m.v(y))$  iff  $\text{frac}(m.v'(x)) = \text{frac}(m.v'(y))$  and  $\text{frac}(m.v(x)) < \text{frac}(m.v(y))$  iff  $\text{frac}(m.v'(x)) < \text{frac}(m.v'(y))$ .

The set of  $\mu$ -regions is denoted by  $\text{reg}_\mu$ . Note that two valuations in the same  $\mu$ -region satisfy in particular exactly the same set of constraints in  $\mathcal{G}(\mu)$ .

### 3 The Timed Controller Synthesis Problem

In this section we define the controller synthesis problem we aim to study. The general framework we will follow is along lines of the one proposed in [AMPS98,DM02] with the necessary generalizations to handle partial observability.

We consider a plant over an alphabet of actions  $\Sigma$  which is partitioned into a set  $\Sigma_C$  of controllable actions and a set  $\Sigma_E$  of environment (or non-controllable) actions. This set of environment actions is further partitioned into observable actions  $\Sigma_E^o$  and unobservable actions  $\Sigma_E^u$ . In a similar way, the set of clocks  $X$  of the plant is constituted of two disjoint sets, the set  $X^r$  of observable (or readable) clocks and the set  $X^u$  of unobservable (or unreadable) clocks. Let us fix  $\hat{\Sigma} = (\Sigma_C, \Sigma_E^o, \Sigma_E^u)$  and  $\hat{X} = (X^r, X^u)$  for the rest of this paper.

A *partially observable plant* (or simply *plant* in the following) over  $(\hat{\Sigma}, \hat{X})$  is a deterministic, finite state, timed transition system  $\mathcal{P}$  over  $(\Sigma, X)$ . Intuitively, we are looking for a controller  $\text{Cont}$  such that the “controlled” plant  $\mathcal{P} \parallel \text{Cont}$  satisfies some given specification. The controller is assumed to have limited power. It can read the clocks of  $X^r$ , and read/reset its own set of clocks which we call  $X_{\text{Cont}}$ . However, it cannot refer to the clocks in  $X^u$ . Concerning the actions, the controller can only observe the actions in  $\Sigma_C \cup \Sigma_E^o$ .

The controller must further satisfy two important requirements. It must be *non-restricting* in the following sense: whenever we have  $\tau \in L^*(\mathcal{P} \parallel \text{Cont})$  and  $\tau.(e, t) \in L^*(\mathcal{P})$  with  $e \in \Sigma_E$ , then we must have  $\tau.(e, t) \in L^*(\mathcal{P} \parallel \text{Cont})$ . It must also be *non-blocking* in that it does not block progress of the plant: whenever  $\tau \in L^*(\mathcal{P} \parallel \text{Cont})$  and  $\tau.(b, t) \in L^*(\mathcal{P})$  with  $b \in \Sigma$ , then there exists  $c \in \Sigma$  and  $t' \in \mathbb{R}_{>0}$  such that  $\tau.(c, t') \in L^*(\mathcal{P} \parallel \text{Cont})$ .

Formally, let  $X_{\text{Cont}}$  be a set of clocks disjoint from  $X$ , and let  $\mu = (X^r \cup X_{\text{Cont}}, m, \max)$  be a given granularity. Then a  $\mu$ -controller for  $\mathcal{P}$  is a deterministic timed transition system  $\text{Cont}$  over  $(\Sigma_C \cup \Sigma_E^o, X^r \cup X_{\text{Cont}})$  of granularity  $\mu$  and with resets only in  $X_{\text{Cont}}$  (i.e. if  $q \xrightarrow{a, g, Y} q'$  in  $\text{Cont}$ , then  $Y \subseteq X_{\text{Cont}}$ ). The behaviour of the “controlled” plant is that of the synchronized product transition system  $\mathcal{P} \parallel \text{Cont}$ , w.r.t. the distributed clock-alphabet  $((\Sigma, X, X), (\Sigma_C \cup \Sigma_E^o, X^r \cup X_{\text{Cont}}, X_{\text{Cont}}))$ . The  $\mu$ -controller  $\text{Cont}$  is a *valid* controller if moreover  $\text{Cont}$  is non-restricting and non-blocking.

We can distinguish several types of specifications that the controlled plant  $\mathcal{P} \parallel \text{Cont}$  may have to satisfy:

- **Internal specifications:** The specification is given by a condition  $\mathcal{F}$  on the states of the plant ( $\mathcal{F}$  can be a Büchi or parity condition, as described in the previous section). The controlled plant  $\mathcal{P} \parallel \text{Cont}$  meets the internal specification  $\mathcal{F}$  whenever

for all timed words  $\sigma$  generated by  $\mathcal{P} \parallel \text{Cont}$ , the unique run of  $\mathcal{P} \parallel \text{Cont}$  on  $\sigma$  satisfies the acceptance condition  $\mathcal{F}$  along the states of the plant. These types of specifications have been considered in [AMPS98], in the framework of fully observable plants.

- **External specifications:** The specification is given by a timed automaton  $\mathcal{S}$  which can represent either the desired (in which case the specification will be said to be positive) or the undesired behaviours (the specification will then be said to be negative). The controlled plant  $\mathcal{P} \parallel \text{Cont}$  meets an external positive specification  $\mathcal{S}$  whenever  $L(\mathcal{P} \parallel \text{Cont}) \subseteq L(\mathcal{S})$ , and the controlled plant  $\mathcal{P} \parallel \text{Cont}$  meets an external negative specification  $\mathcal{S}$  whenever  $L(\mathcal{P} \parallel \text{Cont}) \cap L(\mathcal{S}) = \emptyset$ . In [DM02], such specifications have been studied for fully observable plants.

Note that external specifications are more general than internal specifications, as an internal specification can be transformed to an equivalent external specification by simply using the plant along with the internal specification as a deterministic external specification of desired behaviours.

Depending on whether the resources of the controller are fixed *a priori* or not, we define formally two types of timed controller synthesis problems.

**Definition 1 (Timed controller synthesis problem with non-fixed resources).** *Let  $\mathcal{P}$  be a plant over  $(\hat{\Sigma}, \hat{X})$ . Let  $\mathcal{S}$  be a specification. The timed controller synthesis problem consists in deciding whether there exist some granularity  $\mu = (X_r \cup X_{\text{Cont}}, m, \max)$  (where  $X_{\text{Cont}}$  is disjoint from  $X$ ) and a  $\mu$ -controller  $\text{Cont}$  such that the controlled plant  $(\mathcal{P} \parallel \text{Cont})$  meets the specification  $\mathcal{S}$ .*

**Definition 2 (Timed controller synthesis problem with fixed resources).** *Let  $\mathcal{P}$  be a plant over  $(\hat{\Sigma}, \hat{X})$ . Let  $\mathcal{S}$  be a specification. Let  $\mu = (X_r \cup X_{\text{Cont}}, m, \max)$  be a fixed granularity (where  $X_{\text{Cont}}$  is disjoint from  $X$ ). The timed controller synthesis problem with the fixed resources  $\mu$  consists in deciding whether there exists a  $\mu$ -controller  $\text{Cont}$  such that the controlled plant  $(\mathcal{P} \parallel \text{Cont})$  meets the specification  $\mathcal{S}$ .*

The remainder of the paper is devoted to the study of these two problems, under our general hypothesis of partial observability. We will thus extend both works [AMPS98] and [DM02], where the assumption of full observability hypothesis is made. The results of these works have been summed up in the table on page 181. We study in the next two sections respectively the “Timed controller synthesis problem with non-fixed resources” and the “Timed controller synthesis problem with fixed resources” under the partial observability hypothesis.

## 4 Timed Controller Synthesis with Non-fixed Resources

In the setting of full observation the problem of controller synthesis with non-fixed resources is known to be decidable for deterministic specifications [AMPS98, WTH91, DM02]. In the presence of partial observability however, this problem becomes undecidable:

**Theorem 1.** *The timed controller synthesis problem with non-fixed resources for partially observable plants and deterministic specification is undecidable.*

The proof of this theorem can be done by reduction to the  $\mathbb{Q}$ -universality problem for timed automata, which is known as being undecidable [AD94].

As a simple but important corollary of this theorem, we get the following undecidability result. Note that this implies in particular that the results of [AMPS98] for full observability cannot be extended to partial observability.

**Corollary 1.** *The timed controller synthesis problem with non-fixed resources for partially observable plants and for internal specifications is undecidable.*

## 5 Timed Controller Synthesis with Fixed Resources

We now consider the timed controller synthesis problem with fixed resources. Observe that the problem for deterministic specifications reduces to the problem of non-deterministic specifications which specify undesired behaviours. This is true since a deterministic specification of desired behaviours can be complemented and used as a specification of undesired behaviours. Hence we concentrate on solving the problem for non-deterministic specifications of undesired behaviours.

The technique we use is along the lines of the one used in [DM02]. We first show that the existence of a controller is equivalent to the existence of a winning strategy for player  $C$  (the “control”) in a timed game. We then reduce the existence of a winning strategy for player  $C$  in this timed game to the existence of a winning strategy in a classical untimed game. To take into account partial observability however we need to use a slightly different notion of a timed game from the one in [DM02]. The game graph is done away with, and the players simply play over the alphabet of symbolic actions permitted to the controller. The plant comes into the picture only in describing the winning condition of the timed game.

For rest of the section we fix the following instance of the problem. Let  $\mathcal{P}$  be a plant over  $(\widehat{\Sigma}, \widehat{X})$ ,  $\mathcal{S}$  an arbitrary (*i.e.* we do not assume time determinism) specification of undesired behaviours, and  $\mu = (X_r \cup X_{Cont}, m, max)$  a granularity such that  $X_{Cont} \cap X = \emptyset$ .

### 5.1 Timed Controller Synthesis Problem as a Timed Game

We first define the notion of “timed game” between two players  $C$  (the control) and  $E$  (the environment). As we will see, the timed controller synthesis problem reduces easily to the problem of checking whether player  $C$  has a winning strategy in such a game.

Let  $\nu = (X^r \cup Z, n, max')$  be a granularity such that  $Z \cap X = \emptyset$ . A *timed game*  $\Omega$  based on  $(\widehat{\Sigma}, \widehat{X}, \nu)$  is a triple  $(\Gamma, \mathcal{H}, \mathcal{A})$  where  $\Gamma$  is the symbolic alphabet  $(\Sigma_C \cup \Sigma_E^o) \times atoms_\nu \times 2^Z$ ,  $\mathcal{H}$  is a timed transition system over  $(\Sigma, X)$  and  $\mathcal{A}$  is a timed automaton over  $(\Sigma, X)$ . The game is played between players  $C$  and  $E$ , and a play  $\gamma = u_0 u_1 \dots \in \Gamma^\infty$  is built up as follows. Player  $C$  offers a subset of symbolic actions from  $\Gamma$ , and player  $E$  responds by choosing an action  $u_0$  from that subset. Next, player  $C$  again offers a subset of symbolic actions from  $\Gamma$ , and player  $E$  picks  $u_1$  from it, and

so on. At any point, player  $C$  can offer the empty set of moves, in which case the game ends.

A play in  $\Omega$  is thus a word in  $\Gamma^\infty$ . Whether a play  $\gamma$  is winning for player  $C$  will depend on the “synchronized” symbolic words that  $\gamma$  can produce in conjunction with  $\mathcal{H}$ . Towards this end, for a word  $\gamma \in \Gamma^\infty$ , we define the set of *synchronized words* of  $\gamma$  with respect to  $\mathcal{H}$ , denoted  $\text{synw}_{\mathcal{H}}(\gamma)$ , as follows. Let  $\mathcal{U}_\Gamma$  denote the universal, single-state transition system over  $\Gamma$  which accepts all words of  $\Gamma^\infty$ . Consider the product  $\mathcal{H} \parallel \mathcal{U}_\Gamma$ , w.r.t. the distributed clock-alphabet  $((\Sigma, X, X), (\Sigma_C \cup \Sigma_E^o, X^r \cup Z, Z))$ . Then  $\text{synw}_{\mathcal{H}}(\gamma)$  is defined to be the set of all  $\gamma' \in L_{\text{symp}}(\mathcal{H} \parallel \mathcal{U}_\Gamma)$  such that  $\gamma' \upharpoonright 2 = \gamma$  (recall that the operator  $\upharpoonright$  is defined in Section 2). Note that  $\text{synw}_{\mathcal{H}}(\gamma)$  is a set of finite and infinite words. Also, even if  $\gamma$  is finite,  $\text{synw}_{\mathcal{H}}(\gamma)$  could contain infinite words (there could be a word which after a point, has only actions from  $\Sigma_E^u$ ). Let us denote by  $\text{synw}_{\mathcal{H}}^*(\gamma)$  and  $\text{synw}_{\mathcal{H}}^\omega(\gamma)$ , the set of finite and infinite words in  $\text{synw}_{\mathcal{H}}(\gamma)$ , respectively.

A strategy for player  $C$  is a function  $f : \Gamma^* \rightarrow 2^\Gamma$  such that  $f(\gamma)$  is deterministic, for every  $\gamma \in \Gamma^*$ . Note that  $f(\gamma)$  can be the empty set. We say that a play  $\gamma$  is a play *according to*  $f$  if for every prefix  $\tau.u$  of  $\gamma$ ,  $u \in f(\tau)$ . Let  $\text{plays}_f^\omega(\Omega)$  denote the set of infinite plays and  $\text{plays}_f^*(\Omega)$  denote the set of finite plays played according to  $f$ . We set  $\text{plays}_f(\Omega) = \text{plays}_f^*(\Omega) \cup \text{plays}_f^\omega(\Omega)$ . Note that  $\text{plays}_f(\Omega)$  is prefix-closed.  $f$  will be termed a *finite-state* strategy if there exists a deterministic finite state transition system  $\mathcal{T}$  over  $\Gamma$ , with  $f(\gamma)$  given by the set of actions enabled at  $\text{state}_{\mathcal{T}}(\gamma)$  (defined as the unique – recall that  $\mathcal{T}$  is assumed to be deterministic – state of  $\mathcal{T}$  reachable from the initial state when reading  $\gamma$ ).

For a strategy  $f$  and a finite play  $\gamma \in \text{plays}_f^*(\Omega)$ , let  $\Xi_\gamma = \text{tw}(\text{synw}_{\mathcal{H}}^*(\gamma))$  denote the set of finite timed words generated by the strategy in conjunction with  $\mathcal{H}$  on the finite play  $\gamma$ . We say that  $f$  is *non-restricting* if whenever  $\gamma \in \text{plays}_f^*(\Omega)$ ,  $\sigma \in \Xi_\gamma$  and  $\sigma.(e, t) \in L^*(\mathcal{H})$  with  $e \in \Sigma_E$ , it is the case that  $\sigma.(e, t) \in \Xi_{\gamma'}$  for some  $\gamma' \in \text{plays}_f^*(\Omega)$  such that  $\gamma$  is a prefix of  $\gamma'$ . We say  $f$  is *non-blocking* if whenever we have  $\gamma \in \text{plays}_f^*(\Omega)$ ,  $\sigma \in \Xi_\gamma$ , and  $\sigma.(b, t) \in L^*(\mathcal{H})$  for some  $b \in \Sigma$ , then there is a word  $\sigma.(c, t') \in \Xi_{\gamma'}$ , where  $c \in \Sigma$ ,  $t' \in \mathbb{R}_{>0}$  and  $\gamma'$  is in  $\text{plays}_f^*(\Omega)$  such that  $\gamma$  is a prefix of  $\gamma'$ . We call a strategy *valid* if it is both non-restricting and non-blocking.

A play  $\gamma \in \Gamma^\infty$  in  $\Omega$  is said *winning* for player  $C$  whenever  $\text{tw}(\text{synw}_{\mathcal{H}}^\omega(\gamma)) \cap L(\mathcal{A}) = \emptyset$ . We say that a strategy  $f$  is winning for player  $C$  if all plays according to  $f$  are winning for  $C$  – or equivalently, if

$$\text{tw}(\text{synw}_{\mathcal{H}}^\omega(\text{plays}_f(\Omega))) \cap L(\mathcal{A}) = \emptyset.$$

Let us return to the instance of the controller synthesis problem we have fixed. We define the timed game  $\Omega = (\Delta_{\mu, X_{\text{Cont}}}, \mathcal{P}, \mathcal{S})$  over  $(\widehat{\Sigma}, \widehat{X}, \mu)$  where  $\Delta_{\mu, X_{\text{Cont}}} = (\Sigma_C \cup \Sigma_E^o) \times \text{atoms}_\mu \times 2^{X_{\text{Cont}}}$ . The timed game  $\Omega$  captures our timed controller synthesis problem in the following sense:

**Lemma 1.** *There exists a valid (finite-state)  $\mu$ -controller  $\text{Cont}$  for  $\mathcal{P}$  such that  $(\mathcal{P} \parallel \text{Cont})$  meets the specification  $\mathcal{S}$  iff player  $C$  has a valid (finite-state) winning strategy in the timed game  $\Omega$ .*



## 5.2 Solving a Timed Game

In this section, we reduce the problem of checking whether  $C$  has a valid winning strategy in  $\Omega$  to whether there is a winning strategy for a player in a classical untimed game. Our notion of an untimed game is slightly different from the usual infinite games on finite graphs in the literature (see [McN93]) in that we additionally have a *finitary* winning condition.

An *untimed* game is a tuple  $\Phi = (\Delta, \mathcal{K}, val, \mathcal{B})$  where  $\Delta$  is a finite alphabet,  $\mathcal{K}$  is a finite deterministic transition system over  $\Delta$  (we refer to  $\mathcal{K}$  as the *arena*),  $val$  is a function  $Q \rightarrow 2^{2^\Delta}$  (where  $Q$  is the set of states in  $\mathcal{K}$ ) that identifies a collection of valid sets of moves at each state and  $\mathcal{B}$  is a finite automaton over  $\Delta$  accepting a language  $L_{\text{symb}}(\mathcal{B}) \subseteq \Delta^\infty$ . We require that for any  $q \in Q$ , and  $U \in val(q)$ , all the actions in  $U$  are enabled at  $q$ .

The game is played between two players, player 0 and player 1, as follows. The game starts at the initial state of  $\mathcal{K}$ ; at any state  $q$  in  $\mathcal{K}$ , player 0 picks a set of actions  $U \in val(q)$  and player 1 responds by picking an action  $u$  in  $U$ . The game then continues from the unique  $u$ -successor state of  $q$ . The players hence build up plays which can be finite or infinite words. At any state  $q$ , if  $\emptyset \in val(q)$ , player 0 can choose  $\emptyset$  as its move, and the game stops.

Formally, a strategy for player 0 is a function  $g : \Delta^* \rightarrow 2^\Delta$  such that if  $\alpha \in \Delta^*$  and there is a run of  $\mathcal{K}$  on  $\alpha$  reaching a state  $q$ , then  $g(\alpha) \in val(q)$ . The set of plays according to  $g$ ,  $plays_g(\Phi)$ , is the set of all finite and infinite sequences  $\gamma$  such that for every finite prefix  $\delta.u$  of  $\gamma$ , where  $u \in \Delta$ , we have  $u \in g(\delta)$ . The strategy  $g$  is said to be *winning* for player 0 if  $plays_g(\Phi) \subseteq L_{\text{symb}}(\mathcal{B})$ .

Coming back to timed games, let us fix a granularity  $\nu = (X^r \cup Z, n, max')$  such that  $Z \cap X = \emptyset$ . We also fix a timed game  $\Omega = (\Gamma, \mathcal{H}, \mathcal{A})$  over  $(\hat{\Sigma}, \hat{X}, \nu)$ , where  $\Gamma = (\Sigma_C \cup \Sigma_E^o) \times atoms_\nu \times 2^Z$ , and  $\mathcal{H}$  is a timed transition system over  $(\hat{\Sigma}, \hat{X})$ . In the rest of this section, our aim is to construct an untimed game  $\Phi = (\Gamma, \mathcal{K}, val, \mathcal{B})$  such that player  $C$  has a valid winning strategy in  $\Omega$  iff player 0 has a winning strategy in the untimed game  $\Phi$ .

**Construction of the arena  $\mathcal{K}$ .** Let us first recall the standard region construction used in the analysis of timed automata [AD94]. Consider a timed transition system  $\mathcal{T}$  (resp. a timed automaton  $\mathcal{A}$ ) over a symbolic alphabet  $\Delta$ , with state-space  $Q$ . From the results of [AD94], one can build a transition system  $\mathcal{R}_\mathcal{T}$  (resp. an automaton  $\mathcal{R}_\mathcal{A}$ ) over the symbolic alphabet  $\Delta$ , whose state space is contained in  $Q \times reg_\delta$  (where  $\delta$  is the granularity of  $\Delta$ ) and such that for a symbolic word  $\alpha \in \Delta^*$ , there is a run of  $\mathcal{R}_\mathcal{T}$  (resp.  $\mathcal{R}_\mathcal{A}$ ) on  $\alpha$  iff  $tw(\alpha)$  is nonempty. We call these the *region transition system* and the *region automaton*, respectively.

To construct the arena, we start from the transition system  $\mathcal{H} \parallel \mathcal{U}_\Gamma$  w.r.t. the distributed alphabet  $((\Sigma, X, X), (\Sigma_C \cup \Sigma_E^o, X^r \cup Z, Z))$  (recall the definition of  $\mathcal{U}_\Gamma$  from page 188). Consider the region transition system  $\mathcal{R}$  corresponding to it. This transition system can be viewed as an untimed transition system on the alphabet  $\Delta = \Sigma \times atoms_{\kappa+\mu} \times 2^{X \cup Z}$  where  $\kappa$  is the granularity of  $\mathcal{H}$ .

Let us define now the “projection” of  $\mathcal{R}$  on the atomic alphabet  $\Gamma$ , which has the same set of states and its transitions are obtained by substituting each transition  $s \xrightarrow{a,g,Y} s'$  in  $\mathcal{R}$  by  $s \xrightarrow{(a,g,Y) \upharpoonright 2} s'$  (see page 184 for the definition of the projection  $\upharpoonright$ ). This transition system  $\mathcal{R}'$  is thus a non-deterministic transition system, with  $\epsilon$ -transitions, over the alphabet  $\Gamma$ . By viewing  $\mathcal{R}'$  as a non-deterministic automaton with  $\epsilon$ -transitions on finite words (assuming all states are final), we can determinize it using the usual subset construction, leading to a deterministic automaton  $\mathcal{K}$ . Without loss of generality, we assume  $\mathcal{K}$  is complete, *i.e.* it has a run on every word  $\gamma$  in  $\Gamma^*$ .

We take  $\mathcal{K}$  to be the arena for the untimed game  $\Phi$ . Note that the set of states of  $\mathcal{K}$  is  $2^{H \times \text{reg}_{\kappa+\mu}}$  where  $H$  is the set of states of  $\mathcal{H}$ . Intuitively, if the state reached in  $\mathcal{K}$  on a word  $\gamma \in \Gamma^*$  is  $\{(h_1, r_1), \dots, (h_l, r_l)\}$ , this signifies that on the play  $\gamma$ , there are several (finite) synchronized words w.r.t.  $\mathcal{H}$  and these words end up in one of the states  $h_i$  of  $\mathcal{H}$  along with the clocks in the region  $r_i$ . Using this information, we can now define the valid sets of moves that player 0 is allowed in the untimed game, by referring to the transition system  $\mathcal{R}$  from which  $\mathcal{K}$  was obtained. Let  $s = \{(h_1, r_1), \dots, (h_l, r_l)\}$  be a state of  $\mathcal{K}$ . Then a set of actions  $U \subseteq \Gamma$  is in  $\text{val}(s)$  iff the following conditions hold:

- $U$  is a time deterministic set of actions,
- (*non-restricting*) for each  $(h_i, r_i) \in s$  and  $e \in \Sigma_E^o$ , if  $(h_i, r_i) \xrightarrow{e,g,Y} (h'_i, r'_i)$  is in  $\mathcal{R}$ , then there is an action  $(e, g', Y') \in U$  such that  $g \Rightarrow g'$ ,
- (*non-blocking*) for each  $(h_i, r_i) \in s$ , if there exists some outgoing edge from  $(h_i, r_i)$  in  $\mathcal{R}$ , then there is a transition  $(h_i, r_i) \xrightarrow{a,g,Y} (h'_i, r'_i)$  in  $\mathcal{R}$  and an action  $(a, g', Y') \in U$  such that  $g \Rightarrow g'$ .

Let  $\Phi$  be the untimed game defined above (we leave the winning condition unspecified for the moment). Then it is easy to see that the validity of strategies is preserved across  $\Omega$  and  $\Phi$ :

**Lemma 2.**  $f : \Gamma^* \rightarrow 2^\Gamma$  is a valid strategy for  $C$  in  $\Omega$  iff it is a strategy for player 0 in  $\Phi$ .

**Construction of the winning condition.** It remains now to construct the winning condition  $\mathcal{B}$  for the game  $\Phi$ . The construction is based on the following lemma, whose proof is based on the region automaton for an appropriately defined timed automaton.

**Lemma 3.** *The set of plays in  $\Omega$  which are winning for player  $C$  is regular, *i.e.* there exists an automaton  $\mathcal{B}$  which accepts exactly this set of executions.*

Given a timed game  $\Omega$ , we derive the corresponding untimed arena  $\Phi$  and the winning condition given by the automaton  $\mathcal{B}$  as described above. From Lemma 2 and Lemma 3, the following is immediate:

**Lemma 4.** *Let  $\Omega$  be a timed game and let  $\Phi$  be the corresponding untimed game constructed above. Then, there is a valid (finite-state) winning strategy for player  $C$  in  $\Omega$  iff there is a (finite-state) winning strategy for player 0 in  $\Phi$ .*

Untimed games with winning conditions given as automata on infinite words, are known to be effectively solvable [Tho95]. In our case, we have finitary winning conditions as well. We can handle this by first computing the set of all nodes from which player 1 can force the play to enter a node that is *not* a finitary final state. These states are thus the ones player 0 must not visit during a game. Note that it does not influence the infinite winning condition (because if an infinite path goes through such a state, it has a finite prefix play that is winning for player 1). We can then remove these nodes and solve the resulting game with only the infinitary winning conditions.

### 5.3 The Decision Procedure

For solving an instance of the timed controller synthesis problem, we first find the corresponding timed game  $\Omega$ , as described in section 5.1, and the untimed game  $\Phi$  corresponding to  $\Omega$ , as described in section 5.2. We can then solve  $\Phi$  to obtain a memory-less winning strategy for player 0 whenever he has one. Such a strategy is also a valid winning strategy in  $\Omega$ , which in turn corresponds to a finite state controller which meets the specification  $\mathcal{S}$ . Thus we have:

**Theorem 2.** *The timed controller synthesis problem with fixed resources for partially observable plants and for external negative specifications is decidable and effective.*

The complexity of the decision procedure can be seen to be in time doubly exponential in the size of the instance (see the appendix for details). A point worthy of note here is that this problem, under complete observability is 2EXPTIME-complete [DM02]. Hence, in terms of overall complexity, there is no extra cost paid for partial observability. From the 2EXPTIME lower bound for the complete observation setting, it follows that our problem is also 2EXPTIME-complete.

## 6 Conclusion

The table on page 182 summarizes our results. It is interesting to note that in the timed setting, moving from complete information to partial information preserves only certain decidability results, unlike the situation in a discrete setting where all decidability results carry over [KV97].

The restriction to searching the domain of controllers with limited resources seems to be very useful in the timed controller synthesis problem. In the case of partial observation, it extends the decidability results of complete information, while without this restriction, we get undecidability even for simple specifications. Also, for the complete observation setting, it allows us to have stronger but decidable specification mechanisms [DM02]. We see restricting resources as a very useful restriction which often makes problems decidable and yet remains interesting from the perspective of controller synthesis.

In handling partial observability, for the decidable problems, we have shown results for the general case of external non-deterministic specifications, as done in [DM02] for the complete observation setting. These kinds of specifications are in fact the only truly non-deterministic timed specifications we know for which controller synthesis remains

decidable. The work reported in [FLM02] handles a sub-class of TCTL which can be transformed to automata where clock-resets are in fact deterministic.

There are several variants of the problem studied in this paper which are interesting. First, the assumption that the plant is deterministic is not really a restriction in the partial-observation setting, as one can always make a nondeterministic plant deterministic by adding extra labels to distinguish nondeterministic transitions, and hiding this away from the controller by making it unobservable.

Secondly, we could ask whether we need to control all the resources of the plant. For example, we could ask whether we would still get decidability if we demand that only the number of clocks is fixed, but the granularity of observation of clocks is not fixed. We can show that this still does not suffice and the problem remains undecidable.

## References

- [AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symp. System Structure and Control*, pages 469–474. Elsevier, 1998.
- [BDMP02] P. Bouyer, D. D’Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. Research Report LSV-02-5, LSV, ENS de Cachan, France, 2002.
- [CHR02] F. Cassez, T.A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *Proc. 5th Int. Works. Hybrid Systems: Computation and Control (HSCC’02)*, volume 2289 of *LNCS*, pages 134–148. Springer, 2002.
- [DM02] D. D’Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Proc. 19th Int. Symp. Theoretical Aspects of Computer Science (STACS’02)*, volume 2285 of *LNCS*, pages 571–582. Springer, 2002.
- [FLM02] M. Faella, S. La Torre, and A. Murano. Dense real-time games. In *Proc. 17th Symp. Logic in Computer Science (LICS’02)*, pages 167–176. IEEE Comp. Soc. Press, 2002.
- [KG95] R. Kumar and V.K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic Publishers, 1995.
- [KV97] O. Kupferman and M.Y. Vardi. Synthesis with incomplete information. In *Proc. 2nd Int. Conf. Temporal Logic (ICTL’97)*, pages 91–106. Kluwer, 1997.
- [LW95] F. Lin and W.M. Wonham. Supervisory control of timed discrete-event systems under partial observation. *IEEE Trans. Automatic Control*, 40(3):558–562, 1995.
- [McN93] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.
- [Rei84] J.H. Reif. The complexity of two-player games of incomplete information. *Journal of Computer and System Sciences*, 29(2):274–301, 1984.
- [Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Int. Symp. Theoretical Aspects of Computer Science (STACS’95)*, volume 900 of *LNCS*, pages 1–13. Springer, 1995.
- [Tho02] W. Thomas. Infinite games and verification. In *Proc. 14th Int. Conf. Computer Aided Verification (CAV’02)*, volume 2404 of *LNCS*, pages 58–64. Springer, 2002.
- [WT97] H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Proc. 36th Conf. Decision and Control*, pages 4607–4612. IEEE Comp. Soc. Press, 1997.
- [WTH91] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *Proc. 30th Conf. Decision and Control*, pages 1527–1528. IEEE Comp. Soc. Press, 1991.

# Hybrid Acceleration Using Real Vector Automata<sup>\*</sup>

## (Extended Abstract)

Bernard Boigelot, Frédéric Herbreteau, and Sébastien Jodogne<sup>\*\*</sup>

Université de Liège,  
Institut Montefiore, B28,  
B-4000 Liège, Belgium  
{boigelot,herbreteau,jodogne}@montefiore.ulg.ac.be,  
<http://www.montefiore.ulg.ac.be/~{boigelot,herbreteau,jodogne}/>

**Abstract.** This paper addresses the problem of computing an exact and effective representation of the set of reachable configurations of a linear hybrid automaton. Our solution is based on accelerating the state-space exploration by computing symbolically the repeated effect of control cycles. The computed sets of configurations are represented by *Real Vector Automata (RVA)*, the expressive power of which is beyond that of the first-order additive theory of reals and integers. This approach makes it possible to compute in finite time sets of configurations that cannot be expressed as finite unions of convex sets. The main technical contributions of the paper consist in a powerful sufficient criterion for checking whether a hybrid transformation (i.e., with both discrete and continuous features) can be accelerated, as well as an algorithm for applying such an accelerated transformation on RVA. Our results have been implemented and successfully applied to several case studies, including the well-known leaking gas burner, and a simple communication protocol with timers.

## 1 Introduction

The reachability problem, which consists in checking whether a given set of system configurations can be reached from the initial state, is central to verifying safety properties of computerized systems. A more general form of this problem is to compute the set of all reachable configurations of a given system. Since the reachability set is in general infinite, the result of the computation has to be expressed symbolically rather than explicitly. The goal is to obtain a symbolic representation of this set that is both *exact*, i.e., containing sufficient information for checking without approximation whether a given configuration is reachable

---

<sup>\*</sup> This work was partially funded by a grant of the “Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées”, and by the European IST-FET project ADVANCE (IST-1999-29082).

<sup>\*\*</sup> Research Fellow (“aspirant”) for the Belgian National Fund for Scientific Research (FNRS).

or not, and *effective*, meaning that the safety properties of interest should be easily and efficiently decidable on the basis of the set representation.

This paper addresses the problem of computing the exact reachability set of hybrid automata, which are finite-state machines extended with real variables that obey continuous and discrete dynamical laws [AHH93,ACH<sup>+</sup>95,Hen96]. Hybrid automata have been introduced as natural models for dynamical systems with both discrete and continuous features, such as embedded control programs and timed communication protocols.

A simple way of computing the reachability set of an automaton extended with variables is to explore its state space, starting from the initial configurations, and propagating the reachability information along the transition relation. In the case of hybrid automata, this approach is faced with two difficulties. First, the dynamical laws governing the evolution of real variables are such that a configuration has generally an infinite number of direct successors, due to the continuous nature of time. This problem can be overcome by grouping into *regions* configurations that can be handled together symbolically during the state-space exploration. The second problem is that the set of such reachable regions can generally not be obtained after a finite number of exploration steps.

For restricted classes of hybrid automata such as *Timed Automata* [AD94], it has been shown that exploring a finite region graph is sufficient for deciding the reachability problem. More general classes, such as *Initialized Rectangular Automata* [HKPV98] and *O-Minimal Hybrid Automata* [PLY99] can also be handled by reducing their analysis to the reachability problem for timed automata. The main drawback of this approach is that the size of the region graph grows exponentially with respect to the magnitude of constants that appear in the model. Besides, analyzing more general models such as *Linear Hybrid Automata* leads to region graphs that are inherently not reducible to finite ones.

However, techniques have been developed for exploring infinite structures using a finite amount of resources. In particular, *meta-transitions* [BW94,Boi99] are objects that can be added to the transition relation of an extended automaton in order to speed up its state-space exploration. A meta-transition is usually associated to a cycle in the automaton control graph. During state-space exploration, following a meta-transition leads in one step to all the configurations that could be reached by following the corresponding cycle any possible number of times. Symbolic state-space exploration using meta-transitions thus makes it possible to go arbitrarily deep in the exploration tree by executing a finite number of operations, and can be seen as an *acceleration* method for step-by-step exploration. This method requires a symbolic representation system for the sets of configurations produced by meta-transitions, as well as a decision procedure for checking whether a given cycle can be turned into a meta-transition (in other words, whether one can compute its unbounded repeated effect, as well as effectively perform this computation with represented sets). Representation systems, decision procedures and computation algorithms have been developed for several classes of infinite-state systems including automata extended with

unbounded integer variables [WB95,Boi99] and state machines communicating via unbounded FIFO channels [BG96,BH97,Boi99].

We argue that applying acceleration methods to hybrid automata is essential to being able to analyze exactly systems that cannot be reduced to small finite models. Meta-transitions cannot be straightforwardly applied to hybrid automata, due to the both discrete and continuous nature of these: the data transformation associated to a control cycle of a hybrid automaton is generally non functional (the image of a single configuration by such a transformation may contain more than one configuration), for the time elapsed in each visited location may differ from one run to another. Moreover, one needs a symbolic representation for sets with discrete and continuous features.

In existing work, these drawbacks are avoided in the following ways. In [HPR94,AHH93], *widening* operators are introduced in order to force the result of every acceleration step to be representable as a finite union of convex polyhedra. This approach guarantees the termination of state-space exploration, but is generally only able to produce an upper approximation of the reachability set. In [HL02], an exact acceleration technique is developed for timed automata, using *Difference Bounds Matrices (DBM)*, i.e., restricted convex polyhedra, as a symbolic representation system. This method speeds up the exploration of finite region graphs but, because DBM are generally not expressive enough to represent infinite unions of convex sets, it cannot be directly applied to more general hybrid models. In [CJ99], the authors introduce a cycle acceleration technique for flat automata extended with integer, rational, or real variables. This approach can be applied to the analysis of timed automata, but the considered model is not expressive enough to handle more general hybrid systems. In [AAB00], the analysis of timed automata extended with integer variables and parameters is carried out using *Parametric Difference Bounds Matrices (PDBM)* as representations, as well as an extrapolation technique in order to ensure termination. However, the expressiveness of PDBM is still not sufficient for verifying hybrid systems. In [BBR97], a more expressive representation system, the *Real Vector Automaton (RVA)* is introduced in order to represent sets of vectors with real and integer components. The acceleration method proposed in [BBR97] can only associate meta-transitions to cycles that correspond to functional transformations. This strong restriction prevents this solution from being applied to systems whose timed features make their transition graph inherently non deterministic.

In this paper, we build upon these prior results and develop an acceleration method, based on cyclic meta-transitions, that takes into account both discrete and continuous features of the accelerated transformations, hence the name *hybrid acceleration*. We focus on an exact computation of the reachability set, therefore, owing to the undecidable nature of this problem, our solution takes the form of a partial algorithm, i.e., one that may not terminate. This algorithm relies on Real Vector Automata for representing the computed sets of configurations, which are expressed in the first-order additive theory of integers and reals [BRW98,BJW01]. The technical contributions of the paper consist of

a powerful sufficient criterion for checking whether the repeated iteration of a given control cycle of a linear hybrid automaton produces a set that stays within that theory, as well as an associated algorithm for computing on RVA the effect of such accelerations. Our method has the main advantage of being applicable to systems for which other direct approaches fail. This claim is substantiated by a prototype implementation in the framework of the tool LASH [LASH], that has been successfully applied to several case studies. Those include a direct reachability analysis of the well-known leaking gas burner model [CHR91] and of a simple parametrized communication protocol with timers.

## 2 Linear Hybrid Automata

We use the term *convex linear constraint* to denote a finite conjunction of closed linear constraints with integer coefficients, i.e., a set  $\{\mathbf{x} \in \mathbb{R}^n \mid P\mathbf{x} \leq \mathbf{q}\}$ , with  $P \in \mathbb{Z}^{m \times n}$  and  $\mathbf{q} \in \mathbb{Z}^m$ . The term *linear transformation* denotes a relation of the form  $\{(\mathbf{x}, \mathbf{x}') \in \mathbb{R}^n \times \mathbb{R}^n \mid \mathbf{x}' = A\mathbf{x} + \mathbf{b}\}$ , with  $A \in \mathbb{Z}^{n \times n}$  and  $\mathbf{b} \in \mathbb{Z}^n$ .

**Definition 1.** A Linear Hybrid Automaton (LHA) [AHH93, ACH<sup>+</sup>95, Hen96] is a tuple  $(\mathbf{x}, V, E, v_0, X_0, G, A, I, R)$ , where

- $\mathbf{x}$  is a vector of  $n$  real-valued variables, with  $n > 0$ ;
- $(V, E)$  is a finite directed control graph, the vertices of which are the locations of the automaton. The initial location is  $v_0$ ;
- $X_0$  is an initial region, defined by a convex linear constraint;
- $G$  and  $A$  respectively associate to each edge in  $E$  a guard, which is a convex linear constraint, and an assignment, which is a linear transformation;
- $I$  and  $R$  respectively associate to each location in  $V$  an invariant, which is a convex linear constraint, and a rectangular activity  $(\mathbf{l}, \mathbf{u}) \in \mathbb{Z}^n \times \mathbb{Z}^n$ , which denotes the constraint  $\mathbf{l} \leq \dot{\mathbf{x}} \leq \mathbf{u}$ , where  $\dot{\mathbf{x}}$  is the first derivative of  $\mathbf{x}$ .

The semantics of a LHA  $(\mathbf{x}, V, E, v_0, X_0, G, A, I, R)$  is defined by the transition system  $(Q, Q_0, (\rightarrow_\delta \cup \rightarrow_\tau))$ , where

- $Q = V \times \mathbb{R}^n$  is the set of configurations;
- $Q_0 = \{(v, \mathbf{x}) \in Q \mid v = v_0 \wedge \mathbf{x} \in X_0 \cap I(v_0)\}$  is the set of initial configurations;
- The *discrete-step* transition relation  $\rightarrow_\delta \subseteq Q \times Q$  is such that  $(v, \mathbf{x}) \rightarrow_\delta (v', \mathbf{x}')$  iff there exists  $e \in E$  such that  $e = (v, v')$ ,  $\mathbf{x} \in G(e)$  and  $(\mathbf{x}, \mathbf{x}') \in A(e)$ , and  $\mathbf{x}' \in I(v')$ . Such a transition can also be denoted  $(v, \mathbf{x}) \xrightarrow{e}_\delta (v', \mathbf{x}')$  when one needs to refer explicitly to  $e$ ;
- The *time-step* transition relation  $\rightarrow_\tau \subseteq Q \times Q$  is such that  $(v, \mathbf{x}) \rightarrow_\tau (v', \mathbf{x}')$  iff  $v' = v$ , there exists  $t \in \mathbb{R}_{\geq 0}$  such that  $\mathbf{x} + t\mathbf{l} \leq \mathbf{x}' \leq \mathbf{x} + t\mathbf{u}$ , with  $(\mathbf{l}, \mathbf{u}) = R(v)$ , and  $\mathbf{x}' \in I(v)$ .

Let  $\rightarrow$  denote the relation  $(\rightarrow_\delta \cup \rightarrow_\tau)$ , and let  $\rightarrow^*$  be the reflexive and transitive closure of  $\rightarrow$ . A configuration  $(v', \mathbf{x}') \in Q$  is *reachable from* a configuration  $(v, \mathbf{x}) \in Q$  iff  $(v, \mathbf{x}) \rightarrow^* (v', \mathbf{x}')$ . A configuration is *reachable* iff it is reachable from some configuration in  $Q_0$ . The *reachability set*  $\text{Post}^*(H)$  of a LHA  $H$  is the set of its reachable configurations.



### 3 Symbolic State-Space Exploration

#### 3.1 Introduction

We address the problem of computing the reachability set of a given LHA  $H = (\mathbf{x}, V, E, v_0, X_0, G, A, I, R)$ . This set can generally not be enumerated, since a configuration in the semantic transition graph  $(Q, Q_0, (\rightarrow_\delta \cup \rightarrow_\tau))$  of  $H$  may have uncountably many direct time-step successors, due to the dense nature of time.

One thus uses *symbolic* methods, which basically consist in handling *regions*  $(v, S)$ , with  $v \in V$  and  $S \subseteq \mathbb{R}^n$ , instead of single configurations. The transition relations  $\rightarrow_\delta$  and  $\rightarrow_\tau$  can straightforwardly be extended to regions:

- $(v, S) \rightarrow_\delta (v', S')$  iff there exists  $e \in E$  such that  $e = (v, v')$ , and  $S' = \{\mathbf{x}' \in I(v') \mid (\exists \mathbf{x} \in S)(\mathbf{x} \in G(e) \wedge (\mathbf{x}, \mathbf{x}') \in A(e))\}$ . Such a transition can also be denoted  $(v, S) \xrightarrow{e}_\delta (v', S')$ ;
- $(v, S) \rightarrow_\tau (v', S')$  iff  $v' = v$ , and  $S' = \{\mathbf{x}' \in I(v) \mid (\exists \mathbf{x} \in S)(\exists t \in \mathbb{R}_{\geq 0})(\mathbf{x} + t\mathbf{l} \leq \mathbf{x}' \leq \mathbf{x} + t\mathbf{u})\}$ , with  $(\mathbf{l}, \mathbf{u}) = R(v)$ .

The symbolic exploration of the state space of  $H$  starts from the initial region  $Q_0$ , and computes the reachable regions by adding repeatedly to the current set the regions that can be reached by following the relations  $\rightarrow_\delta$  and  $\rightarrow_\tau$ . The computation ends when a fixpoint is reached.

Termination of state-space exploration is clearly not guaranteed, due to the undecidability of the reachability problem for LHA [ACH<sup>+</sup>95, Hen96]. However, the simple algorithm outlined above can substantially be improved by applying *acceleration*, the purpose of which is to explore an infinite number of reachable regions in finite time. Acceleration methods are introduced in Section 3.3, and are then applied to LHA in Section 4.

Now, in order to be able to carry out algorithmically symbolic state-space exploration, one needs a *symbolic representation* for the regions that must be manipulated. This representation has to satisfy some requirements. First, it must be closed under the set operations that need to be performed during exploration. These include the classical set-theory operations  $\cup, \subseteq, \times, \dots$ , as well as computing the successors of regions by the discrete-step and time-step relations. Second, one should be able to carry out the acceleration operations with represented sets. Finally, the system properties that are to be checked must be easily decidable from the representation of the computed reachability set.

The traditional symbolic representations used in the framework of hybrid automata analysis are the finite unions of convex polyhedra [HPR94, HH94], and the *Difference Bounds Matrices (DBM)* [Dil89]. These representations are not able to handle sets that cannot be expressed as finite unions of convex polyhedra, such as the reachability set of the leaking gas burner [CHR91]. In Section 3.2, we recall powerful representations for sets of real vectors, the *Real Vector Automata*, that have the advantage of being much more expressive than geometrical representations, and have good properties that allow an efficient manipulation of represented sets.

### 3.2 Real Vector Automata

Consider an integer base  $r > 1$ . Using the positional number system in base  $r$ , a positive real number  $x$  can be written as an infinite word over the alphabet  $\Sigma = \{0, 1, \dots, r-1, \star\}$ , where “ $\star$ ” denotes a separator between the integer and the fractional parts. For example,  $5/2$  can be written as  $10 \star 1(0)^\omega$ , where “ $\omega$ ” denotes infinite repetition. In a similar way, negative numbers and real vectors with a fixed dimension  $n$  can also be encoded in base  $r$  as infinite words over the finite alphabet  $\Sigma$  [BBR97,BRW98,BJW01].

Given a set  $S \subseteq \mathbb{R}^n$ , let  $L(S) \subseteq \Sigma^\omega$  denote the language of all the encodings of all the vectors in  $S$ . If  $L(S)$  is  $\omega$ -rational, then it is accepted by a Büchi automaton  $A$ , which is said to be a *Real Vector Automaton (RVA)* that *represents* the set  $S$ .

It is known that the sets of real vectors that can be represented by RVA are those that are definable in a base-dependent extension of the first-order theory  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  [BRW98]. RVA are thus expressive enough to handle linear constraints over real and integer variables, as well as periodicities, and are closed under Boolean operators and first-order quantifiers. Moreover, it has been shown that staying within  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  makes it possible to avoid the use of the complex and costly algorithms that are usually required for handling infinite-words automata [BJW01]. Moreover, RVA admit a normal form [Löd01], which speeds up set comparisons and prevents the representations from becoming unnecessarily large. An implementation of RVA restricted to  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  is available in the framework of the LASH toolset [LASH].

### 3.3 Acceleration Methods

The idea behind acceleration is to capture the effect of selected *cycles* in the control graph  $(V, E)$  of the LHA  $H$  being analyzed. Let  $\sigma = e_1; e_2; \dots; e_p$  be such a cycle, where for each  $i \in [1, \dots, p]$ ,  $v_i$  is the origin of  $e_i$ . The transition  $e_i$  has the destination  $v_{i+1}$  if  $i < p$ , and  $v_1$  if  $i = p$ . The transformation  $\theta : 2^{\mathbb{R}^n} \rightarrow 2^{\mathbb{R}^n}$  associated to  $\sigma$  is such that  $\theta(S) = S'$  iff there exist  $S_1, S'_1, S_2, S'_2, \dots, S'_{p-1}, S_p \subseteq \mathbb{R}^n$  such that  $S_1 = S$  and  $S_p = S'$ , and for each  $i \in [1, \dots, p-1]$ ,  $(v_i, S_i) \rightarrow_\tau (v_i, S'_i)$  and  $(v_i, S'_i) \xrightarrow{e_i}_\delta (v_{i+1}, S_{i+1})$ .

The *meta-transition* [WB95,BBR97,Boi99] corresponding to  $\sigma$  is defined as a relation  $\rightarrow_\sigma$  over the regions of  $H$  such that  $(v, S) \rightarrow_\sigma (v', S')$  iff  $v = v' = v_1$  and  $S' = \theta^*(S)$ , where  $\theta^* = \bigcup_{i \geq 0} \theta^i$ . Clearly, meta-transitions preserve reachability, in the sense that  $(v', S')$  is reachable if  $(v, S)$  is reachable. They can thus be added to the semantic transition system of an LHA in order to speed up its state-space exploration. A meta-transition is able to produce in one step regions that could only be obtained after an unbounded number of iterations of the corresponding cycle. Meta-transitions thus provide an acceleration strategy that makes it possible to explore infinite region graphs (though not all of them) in finite time. Meta-transitions can either be selected manually, or discovered by automatic or semi-automatic methods [Boi99].

The use of meta-transitions requires a decision procedure for checking whether the closure  $\theta^*$  of a given transformation  $\theta$  can effectively be constructed, and whether the image by this closure of a set of configurations can be computed within the symbolic representation used during the analysis. Those problems are tackled in Section 4, in the case of LHA analyzed using Real Vector Automata.

## 4 Hybrid Acceleration

### 4.1 Linear Hybrid Transformations

We establish now the general form of the data transformations on which acceleration can be applied. We first consider the case of a path  $\sigma$  performing a time step at some location  $v$  followed by a discrete step along an edge  $e$  from  $v$  to  $v'$ . The data transformation  $\theta$  associated to  $\sigma$  is such that

$$\theta(S) = \{\mathbf{x}' \in \mathbb{R}^n \mid (\exists \mathbf{x} \in S)(\exists \mathbf{x}'' \in \mathbb{R}^n)(\exists t \in \mathbb{R}_{\geq 0})(\mathbf{x} + t\mathbf{l} \leq \mathbf{x}'' \leq \mathbf{x} + t\mathbf{u} \wedge \mathbf{x}'' \in I(v) \wedge \mathbf{x}'' \in G(e) \wedge (\mathbf{x}'', \mathbf{x}') \in A(e) \wedge \mathbf{x}' \in I(v'))\},$$

where  $(\mathbf{l}, \mathbf{u}) \in R(v)$ .

By Definition 1, the constraints  $I(v)$ ,  $G(e)$ ,  $A(e)$ , and  $I(v')$  are systems of linear equalities and inequalities. The previous formula can thus be rewritten as  $\theta(S) = \{\mathbf{x}' \in \mathbb{R}^n \mid (\exists \mathbf{x} \in S)(\exists \mathbf{x}'' \in \mathbb{R}^n)(\exists t \in \mathbb{R}_{\geq 0})\varphi(\mathbf{x}, \mathbf{x}', \mathbf{x}'', t)\}$ , where  $\varphi$  is a conjunction of linear inequalities, in other words a closed convex polyhedron in  $\mathbb{R}^{3n+1}$ . Such polyhedra are closed under projection. One can therefore project out the quantified variables  $\mathbf{x}''$  and  $t$ , which yields  $\theta(S) = \{\mathbf{x}' \in \mathbb{R}^n \mid (\exists \mathbf{x} \in S)\varphi'(\mathbf{x}, \mathbf{x}')\}$ , where  $\varphi'$  is a conjunction of linear inequalities. This prompts the following definition.

**Definition 2.** A Linear Hybrid Transformation (LHT) is a transformation of the form

$$\theta : 2^{\mathbb{R}^n} \rightarrow 2^{\mathbb{R}^n} : S \mapsto \left\{ \mathbf{x}' \in \mathbb{R}^n \mid (\exists \mathbf{x} \in S) \left( P \begin{bmatrix} \mathbf{x} \\ \mathbf{x}' \end{bmatrix} \leq \mathbf{q} \right) \right\},$$

with  $n > 0$ ,  $P \in \mathbb{Z}^{m \times 2n}$ ,  $\mathbf{q} \in \mathbb{Z}^m$  and  $m \geq 0$ .

Notice that a LHT is entirely defined by the linear system induced by  $P$  and  $\mathbf{q}$ . In the sequel, for simplicity sake, we denote such a LHT by the pair  $(P, \mathbf{q})$ .

We have just established that data transformations associated to a time step followed by a discrete step can be expressed as LHT. Thanks to the following result, the transformations corresponding to arbitrary control paths of LHA can be described by LHT as well.

**Theorem 1.** Linear Hybrid Transformations are closed under composition.

*Proof Sketch.* Immediate by quantifying away the intermediate variables.  $\square$

Note that the image by a LHT  $(P, \mathbf{q})$  of a single vector  $\mathbf{x}_0 \in \mathbb{R}^n$  is the set  $\{\mathbf{x} \in \mathbb{R}^n \mid P''\mathbf{x} \leq \mathbf{q} - P'\mathbf{x}_0\}$ , where  $[P'; P''] = P$  with  $P', P'' \in \mathbb{Z}^{m \times n}$ , which defines a closed convex polyhedron. Each constraint of this polyhedron has the form  $\mathbf{p}'' \cdot \mathbf{x} \leq \mathbf{q} - \mathbf{p}' \cdot \mathbf{x}_0$ , hence has coefficients that are independent from the initial vector  $\mathbf{x}_0$ , and an additive term that depends linearly on  $\mathbf{x}_0$ .

## 4.2 Iterating Transformations

We now address the problem of computing within  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  the closure of a transformation  $\theta$  defined by a Linear Hybrid Transformation  $(P, \mathbf{q})$ .

Clearly, there exist transformations  $\theta$  and sets  $S \subseteq \mathbb{R}^n$  definable in  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  such that  $\theta^*(S)$  does not belong to that theory<sup>1</sup>. Determining exactly whether  $\theta^*(S)$  is definable in  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  for all definable sets  $S$  is a hard problem. Indeed, a solution to this problem would also cover the simpler case of guarded discrete linear transformations, for which no decision procedure is known [Boi99]. A realistic goal is thus to obtain a *sufficient* criterion on  $\theta$ , provided that it is general enough to handle non trivial hybrid accelerations, and that the closure of every transformation that satisfies the criterion can be effectively computed.

A first natural restriction consists of requiring that the  $k$ -th image by  $\theta$ , denoted  $\theta^k(S)$ , of a set  $S$  definable in  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  is definable in that theory in terms of elements  $\mathbf{x}_0$  of  $S$  and  $k$ . The set  $\theta^*(S)$  could then be computed by existentially quantifying  $\theta^k(S)$  over  $k \geq 0$  and  $\mathbf{x}_0 \in S$ .

Consider an arbitrary  $\mathbf{x}_0 \in S$  and a fixed value  $k > 0$ . Since by Theorem 1, the transformation  $\theta^k$  can be expressed as a LHT, the set  $\theta^k(\{\mathbf{x}_0\})$  is a closed convex polyhedron  $\Pi_k$ . Such a polyhedron is uniquely characterized by its set of *vertices* and *extremal rays* [Wey50]. We now study the evolution of the vertices and rays of  $\Pi_k$ , for all  $k > 0$ , as functions of  $\mathbf{x}_0$  and  $k$ .

We have  $\Pi_1 = \{\mathbf{x} \in \mathbb{R}^n \mid P''\mathbf{x} \leq \mathbf{q} - P'\mathbf{x}_0\}$ , with  $[P'; P''] = P$ . Let  $\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_p$  be the vertices and rays of  $\Pi_1$ . Each  $\mathbf{r}_i$  is uniquely characterized as the intersection of a particular subset of constraint boundaries. In other words,  $\mathbf{r}_i$  is the only solution to an equation of the form  $P_i\mathbf{r}_i = \mathbf{q}_i(\mathbf{x}_0)$ , where  $P_i$  does not depend on  $\mathbf{x}_0$ , and  $\mathbf{q}_i(\mathbf{x}_0)$  depends linearly on  $\mathbf{x}_0$ . From this property, we define the *trajectory* of  $\mathbf{r}_i$  with respect to  $\theta$  as the infinite sequence  $\mathbf{r}_i^0, \mathbf{r}_i^1, \mathbf{r}_i^2, \dots$  where  $\mathbf{r}_i^0 = \mathbf{r}_i$ , and for each  $j > 0$ ,  $P_i\mathbf{r}_i^j = \mathbf{q}_i(\mathbf{r}_i^{j-1})$ .

It is known [Wei99] that the sets of discrete values that are definable in  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  are *ultimately periodic*, i.e., they can be expressed as a finite union of sets of the form  $\{\mathbf{a} + j\mathbf{b} \mid j \in \mathbb{N}\}$ . Thus, a natural restriction on  $\theta$  is to require that each vertex or ray of the image polyhedron  $\theta$  follows a periodic trajectory. Unfortunately, such a criterion would be rather costly to check explicitly, for the number of vertices and rays of a  $n$ -dimensional polyhedron may grow exponentially in  $n$ . However, since each vertex or ray is an intersection of constraint boundaries which take the form of  $(n - 1)$ -planes, its trajectory is always periodic whenever these hyperplanes follow periodic trajectories themselves. It is therefore sufficient to impose the periodicity restriction on the *linear constraints* that characterize  $\theta$  rather than on the vertices and rays of its image polyhedron. We are now ready to formally characterize the Linear Hybrid Transformations on which acceleration can be applied.

**Definition 3.** Let  $\theta = (P, \mathbf{q})$ , and let  $C$  be a constraint of  $\theta$ , i.e., a row  $\mathbf{p}''.\mathbf{x}' \leq \mathbf{q} - \mathbf{p}'.\mathbf{x}$  of the underlying linear system of  $\theta$ . Let  $\theta_C$  be the LHT induced by the boundary of  $C$ , i.e., transforming  $\mathbf{x}$  into  $\mathbf{x}'$  such that  $\mathbf{p}''.\mathbf{x}' = \mathbf{q} - \mathbf{p}'.\mathbf{x}$ .

<sup>1</sup> Consider for instance the transformation  $x \mapsto 2x$  with  $S = \{1\}$ .

For each  $d \in \mathbb{R}$ , let  $\Gamma_{(C,d)}$  denote the set  $\{\mathbf{x}' \in \mathbb{R}^n \mid \mathbf{p}'' \cdot \mathbf{x}' = d\}$  (that is, the general form of the image by  $\theta_C$  of a single vector).

The transformation  $\theta_C$  is periodic if  $\mathbf{p}'' = \mathbf{0}$  (in which case the image of every vector is the empty set or the universal set), or if the following conditions are both satisfied:

- For every  $d \in \mathbb{R}$ , there exists a single  $d' \in \mathbb{R}$  such that  $\theta_C(\Gamma_{(C,d)}) = \Gamma_{(C,d')}$  (this condition ensures that the image of the  $(n-1)$ -plane  $\Gamma_{(C,d)}$  is a  $(n-1)$ -plane parallel to it);
- Let  $d_0, d_1, \dots \in \mathbb{R}$  be such that for every  $j > 0$ ,  $\theta_C(\Gamma_{(C,d_j)}) = \Gamma_{(C,d_{j-1})}$ . The sequence  $d_0, d_1, \dots$  is such that for every  $j > 0$ ,  $d_j - d_{j-1} = d_{j+1} - d_j$  (in other words, the sequence must be an arithmetic progression).

**Definition 4.** A LHT is periodic if the transformations induced by all its underlying constraints are periodic.

The periodicity of a LHT can be checked easily thanks to the following result.

**Theorem 2.** Let  $\theta_C$  be a LHT transforming  $\mathbf{x}$  into  $\mathbf{x}'$  such that  $\mathbf{p}'' \cdot \mathbf{x}' = q - \mathbf{p}' \cdot \mathbf{x}$ . This transformation is periodic iff  $\mathbf{p}'' = \mathbf{0}$  or  $\mathbf{p}' = \lambda \mathbf{p}''$ , with  $\lambda \in \{0, -1\}$ .

*Proof Sketch.* If  $\mathbf{p}'' = \mathbf{0}$ , the result is immediate. If  $\mathbf{p}'' \neq \mathbf{0}$ , then the image by  $\theta_C$  of a set  $\Gamma_{(C,d_i)}$  is a  $(n-1)$ -plane if and only if  $\mathbf{p}'$  and  $\mathbf{p}''$  are colinear (otherwise, the image is  $\mathbb{R}^n$ ). Let  $\lambda \in \mathbb{R}$  be such that  $\mathbf{p}' = \lambda \mathbf{p}''$ . The periodicity condition on a sequence  $d_0, d_1, \dots$  constructed from an arbitrary  $d_0 \in \mathbb{R}$  can only be fulfilled if  $\lambda \in \{0, -1\}$ . The details of the last step are omitted from this proof sketch.  $\square$

The previous theorem leads to a simple characterization of periodic transformations.

**Theorem 3.** A LHT  $(P, \mathbf{q})$  is periodic if and only if its underlying linear system is only composed of constraints of the form  $\mathbf{p} \cdot \mathbf{x} \leq q$ ,  $\mathbf{p} \cdot \mathbf{x}' \leq q$ , and  $\mathbf{p} \cdot (\mathbf{x}' - \mathbf{x}) \leq q$ .

### 4.3 Image Computation

In this section, we address the problem of computing effectively  $\theta^*(S)$ , given a periodic LHT  $\theta = (P, \mathbf{q})$  and a set  $S$  represented by a RVA.

Since  $\theta$  is periodic, its underlying linear system can be decomposed into  $P_0 \mathbf{x} \leq \mathbf{q}_0 \wedge P_1(\mathbf{x}' - \mathbf{x}) \leq \mathbf{q}_1 \wedge P_2 \mathbf{x}' \leq \mathbf{q}_2$ , with  $P_0 \in \mathbb{Z}^{m_0 \times n}$ ,  $P_1 \in \mathbb{Z}^{m_1 \times n}$ ,  $P_2 \in \mathbb{Z}^{m_2 \times n}$ ,  $\mathbf{q}_0 \in \mathbb{Z}^{m_0}$ ,  $\mathbf{q}_1 \in \mathbb{Z}^{m_1}$ ,  $\mathbf{q}_2 \in \mathbb{Z}^{m_2}$ , and  $m_0, m_1, m_2 \in \mathbb{N}$ . We first study transformations for which  $m_0 = m_2 = 0$ .

**Theorem 4.** Let  $\theta_1$  be the LHT characterized by the linear system  $P_1(\mathbf{x}' - \mathbf{x}) \leq \mathbf{q}_1$ . The LHT induced by the system  $P_1(\mathbf{x}' - \mathbf{x}) \leq k \mathbf{q}_1$ , with  $k > 0$ , is equal to the  $k$ -th power  $\theta_1^k$  of  $\theta_1$ .

*Proof Sketch.* The proof is based on a convexity argument, showing that the trajectory of a vector by the transformation  $\theta_1$  can always be constrained to follow a straight line. The details are omitted from this extended abstract due to space requirements.  $\square$

Let now consider a periodic LHT  $\theta$  for which  $m_0$  and  $m_2$  are not necessarily equal to zero. Let  $P_0\mathbf{x} \leq \mathbf{q}_0 \wedge P_1(\mathbf{x}' - \mathbf{x}) \leq \mathbf{q}_1 \wedge P_2\mathbf{x}' \leq \mathbf{q}_2$  be its underlying linear system,  $\theta_1$  be the LHT induced by  $P_1(\mathbf{x}' - \mathbf{x}) \leq \mathbf{q}_1$ , and let  $C_\theta = \{\mathbf{x} \in \mathbb{R}^n \mid P_0\mathbf{x} \leq \mathbf{q}_0 \wedge P_2\mathbf{x} \leq \mathbf{q}_2\}$ .

**Theorem 5.** *Periodic  $\theta$  are such that for every  $S \subseteq \mathbb{R}^n$  and  $k \geq 2$ ,  $\theta^k(S) = \theta(\theta_1^{k-2}(\theta(S) \cap C_\theta) \cap C_\theta)$ .*

*Proof Sketch.* The proof is based on a convexity argument.  $\square$

Theorems 4 and 5 give a definition within  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$  of  $\theta^k(S)$  as a function of  $k$  and  $S$ , for all periodic LHT  $\theta$ . The expression of  $\theta^k(S)$  can be turned into an algorithm for applying  $\theta^k$  to sets represented by RVA. By quantifying  $k$  over  $\mathbb{N}$ , this algorithm can be used for computing  $\theta^*(S)$  given  $\theta$  and a representation of  $S$ .

#### 4.4 Reduction Step

The applicability of the acceleration method developed in Sections 4.2 and 4.3 is limited by the following observation: there exist LHT  $\theta$  that do not satisfy the hypotheses of Theorem 3, but such that the sequence of sets  $\theta(\mathbf{x}), \theta^2(\mathbf{x}), \theta^3(\mathbf{x}), \dots$  induced by any vector  $\mathbf{x}$  has a periodic structure. This is mainly due to the fact that the *range* of such  $\theta$ , which is the smallest set covering the image by  $\theta$  of every vector in  $\mathbb{R}^n$ , may have a dimension less than  $n$ . In such a case, since the behavior of  $\theta$  outside of its range has no influence on its closure, it is sufficient to study the periodicity of  $\theta$  in a subspace smaller than  $\mathbb{R}^n$ .

We solve this problem by reducing LHT in the following way, before computing their closure. The range  $\theta(\mathbb{R}^n)$  of  $\theta$  is the projection onto  $\mathbf{x}'$  of its underlying linear system, expressed over the variable vectors  $\mathbf{x}$  and  $\mathbf{x}'$ . It hence takes the form of a closed convex polyhedron  $\Pi$ . The largest vector subspace that includes  $\Pi$  can be obtained by first translating  $\Pi$  by some vector  $\mathbf{v}$  in order to make it cover the origin vector  $\mathbf{0}$ , and then extracting from the resulting polyhedron a finite vector basis  $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_{n'}$ , i.e., a maximal set of linearly independent vectors (this can be done automatically). Then, if  $n' < n$ , we change variables from  $\mathbf{x} \in \mathbb{R}^n$  to  $\mathbf{y} \in \mathbb{R}^{n'}$  such that  $\mathbf{x} = U\mathbf{y} + \mathbf{v}$ , where  $U = [\mathbf{u}_1; \mathbf{u}_2; \dots; \mathbf{u}_{n'}]$ . This operation transforms  $\theta = (P, \mathbf{q})$  into the LHT

$$\theta' = \left( P \begin{bmatrix} U & 0 \\ 0 & U \end{bmatrix}, \mathbf{q} - P \begin{bmatrix} \mathbf{v} \\ \mathbf{v} \end{bmatrix} \right),$$

the periodic nature of which can then be checked.

**Theorem 6.** *Let  $\mathbf{x} \mapsto \mathbf{y} : \mathbf{x} = U\mathbf{y} + \mathbf{v}$  be a variable change operation transforming  $\theta$  into  $\theta'$ . For every  $S \subseteq \mathbb{R}^n$ , we have  $\theta^*(S) = S \cup (U((\theta')^*(S')) + \mathbf{v})$ , where  $\theta(S) = US' + \mathbf{v}$ .*

*Proof Sketch.* Immediate by simple algebra.  $\square$

It is worth mentioning that the transformation expressed by the previous theorem can be carried out within  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ .

## 5 Experiments

The acceleration technique developed in this paper has been implemented in the LASH toolset [LASH] and applied to two simple case studies.

The first experiment consisted in analyzing the *leaking gas burner* described in [CHR91,ACH<sup>+</sup>95]. We computed the reachability set of this model, after creating a meta-transition corresponding to the only simple cycle, and reducing its associated transformation to a subspace of dimension 2 (by applying the automatic procedure outlined in Section 4.4). The result of our analysis takes the form of a RVA that can be used for deciding quickly any safety property expressed in  $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ .

Our second case study tackled the analysis of a generalization of the Alternating Bit Protocol [BSW69] to message ranges between 0 and  $N - 1$ , where  $N \geq 2$  is an unbounded parameter. Moreover, the sender and receiver processes use timeouts to guess message losses, then reemitting until the expected acknowledgement is received.

The verification of this protocol has already been successfully achieved in previous work, however using techniques that required to abstract (at least) its temporal specifications. Using our acceleration method, we were able to compute exactly and in a direct way the reachability set of the full protocol, as well as to verify that the message sequence could not be broken.

## 6 Conclusions

This paper introduces a new acceleration method for computing the reachability set of Linear Hybrid Automata. Hybrid acceleration takes advantage of the periodicity of both discrete and continuous transitions, and favors the exact computation of the reachability set rather than its termination, as opposed to the widely spread approximative methods based on widening operators [HPR94,HH94]. Our method has been successfully applied to case studies for which other direct approaches fail, such as the analysis of the leaking gas burner model described in [CHR91].

Our work can be viewed as an extension of [BBR97], where only cycles that behave deterministically (w.r.t. to continuous steps) could be accelerated. It uses a more expressive model, and a more expressive symbolic representation than [CJ99,AAB00,HL02]. Furthermore, using hybrid acceleration with RVA, one can overcome the recently pointed out limitations of the representation of sets using Difference Bounds Matrices [Bou03].

Finally, it should be pointed out that Real Vector Automata provide a symbolic representation that is expressive enough to handle hybrid acceleration, and



efficient enough to handle nontrivial case studies. This motivates the integration of finite-state representations in verification tools for hybrid automata.

## References

- [AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic Techniques for Parametric Reasoning about Counters and Clock Systems. In *Proc. of the 12th International Conference on Computer-Aided Verification (CAV)*, number 1855 in *Lecture Notes in Computer Science*, pages 419–434, Chicago, USA, July 2000. Springer-Verlag.
- [ACH<sup>+</sup>95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.
- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994.
- [AHH93] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *Proc. 14th annual IEEE Real-Time Systems Symposium*, pages 2–11, 1993.
- [BBR97] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In *Proc. of the 9th International Conference on Computer-Aided Verification (CAV)*, number 1254 in *Lecture Notes in Computer Science*, pages 167–177, Haifa, Israël, June 1997. Springer-Verlag.
- [BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of the 8th International Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12, New Brunswick, NJ, USA, July/August 1996. Springer-Verlag.
- [BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO channel systems with nonregular sets of configurations. In *Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 560–570, Bologna, Italy, 1997. Springer-Verlag.
- [BJW01] B. Boigelot, S. Jodogne, and P. Wolper. On the use of weak automata for deciding linear arithmetic with integer and real variables. In *Proc. International Joint Conference on Automated Reasoning (IJCAR)*, volume 2083 of *Lecture Notes in Computer Science*, pages 611–625, Sienna, Italy, June 2001.
- [Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Collection des publications de la Faculté des Sciences Appliquées de l’Université de Liège, Liège, Belgium, 1999.
- [Bou03] P. Bouyer. Untameable timed automata! In *Proc. of 20th Ann. Symp. Theoretical Aspects of Computer Science (STACS)*, *Lecture Notes in Computer Science*, Berlin, Germany, February 2003. Springer-Verlag.
- [BRW98] B. Boigelot, S. Rassart, and P. Wolper. On the expressiveness of real and integer arithmetic automata. In *Proc. of the 25th Colloquium on Automata, Programming, and Languages (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 152–163. Springer-Verlag, July 1998.



- [BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, May 1969.
- [BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In D. L. Dill, editor, *Proc. of the 6th Int. Conf. on Computer Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Stanford, USA, June 1994. Springer-Verlag.
- [CHR91] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.
- [CJ99] H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *Proc. of the 10th Int. Conf. Concurrency Theory (CONCUR)*, volume 1664 of *Lecture Notes in Computer Science*, pages 242–257, Eindhoven, The Netherlands, August 1999. Springer-Verlag.
- [Dil89] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. of Automatic Verification Methods for Finite-State Systems*, number 407 in *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.
- [Hen96] T. A. Henzinger. The theory of hybrid automata. In *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.
- [HH94] T. A. Henzinger and P.-H. Ho. Model checking strategies for linear hybrid systems. In *Proc. of Workshop on Formalisms for Representing and Reasoning about Time*, May 1994.
- [HKPV98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What’s decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.
- [HL02] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. *Electronic Notes in Theoretical Computer Science*, 65(6), April 2002.
- [HPR94] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In *Proc. of the Int. Symposium on Static Analysis*, volume 818 of *Lecture Notes in Computer Science*, pages 223–237. Springer-Verlag, 1994.
- [LASH] The Liège Automata-based Symbolic Handler (LASH). Available at : <http://www.montefiore.ulg.ac.be/~boigelot/research/lash/>.
- [Löd01] C. Löding. Efficient minimization of deterministic weak  $\omega$ -automata. *Information Processing Letters*, 79(3):105–109, 2001.
- [PLY99] G. J. Pappas, G. Lafferriere, and S. Yovine. A new class of decidable hybrid systems. In *Proc. of Hybrid Systems: Computation and Control (HSCC)*, volume 1569 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1999.
- [WB95] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. of the 2nd Int. Symp. on Static Analysis (SAS)*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, 1995.
- [Wei99] V. Weispfenning. Mixed real-integer linear quantifier elimination. In *Proc. of the 1999 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 129–136, New York, July 1999. ACM Press.
- [Wey50] H. Weyl. The elementary theory of convex polyhedra. *Annals of Math. Study*, 24, 1950.

# Abstraction and BDDs Complement SAT-Based BMC in *DiVer*

Aarti Gupta<sup>1</sup>, Malay Ganai<sup>1</sup>, Chao Wang<sup>2</sup>, Zijiang Yang<sup>1</sup>, Pranav Ashar<sup>1</sup>

<sup>1</sup> NEC Laboratories America, Princeton, NJ, U.S.A.

<sup>2</sup> University of Colorado, Boulder, CO, U.S.A.

**Abstract.** Bounded Model Checking (BMC) based on Boolean Satisfiability (SAT) procedures has recently gained popularity for finding bugs in large designs. However, due to its incompleteness, there is a need to perform deeper searches for counterexamples, or a proof by induction where possible. The *DiVer* verification platform uses abstraction and BDDs to complement BMC in the quest for completeness. We demonstrate the effectiveness of our approach in practice on industrial designs.

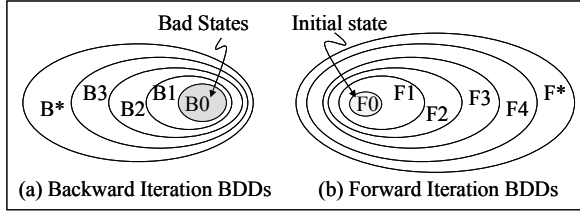
## 1 Introduction

Bounded Model Checking (BMC) [1] has recently become popular for finding bugs in large designs. In this paper, we describe the use of abstraction and BDDs to complement SAT-based BMC in our verification platform called *DiVer*, which includes backend engines for BDD-based symbolic model checking techniques [2, 3], and for Boolean Satisfiability (SAT) [4]. While BDD-based model checking provides a complete proof method, it works only on small models. In contrast, SAT-based BMC can handle much larger models, but it is incomplete in practice. Conservative abstractions that over-approximate the paths in a given model, are key in providing a link between the two. In particular, we use *conservative approximations of certain state sets, computed as BDDs, as additional constraints in BMC* for various applications such as search for counterexamples [1], and proofs by induction [5]. Recent related efforts have used BDD-based enlarged targets [6], and BDD-based approximate reachability state sets [7], in searching for counterexamples using BMC.

## 2 Generation of BDD Constraints for BMC

Since the BDD constraints for BMC are required to be over-approximations, we obtain “existential” abstractions of the design [8], by considering some latches in the concrete design as pseudo-primary inputs. Good candidates are – latches farther in

the dependency closure of the property signals identified by localization techniques [9], peripheral/pipelining latches that do not contribute to the sequential core of the design [10, 11]. Given a conservative abstract model, and a correctness property, we use exact or approximate symbolic model checking techniques [3, 8] to generate the BDD constraints. For simple safety properties, we store the union of the state sets computed iteratively at each step by the pre-image operation, backwards from the bad states until convergence, as shown in Figure 1(a). We also store the union of the state sets in each step of the forward reachability analysis, starting from the initial state until convergence, as shown in Figure 1(b).



**Fig. 1.** Generation of BDD Constraints

We convert each BDD to either a gate-level circuit or a CNF formula, where each internal BDD node is regarded as a multiplexor controlled by that node's variable. The size of the resulting circuit or CNF formula is linear in the size of the BDD, due to introduction of extra CNF variables for each BDD node. We use reordering heuristics as well as over-approximation methods to keep down the BDD sizes.

### 3 Use of BDD Constraints in BMC Applications

The ability of BMC to perform a deeper search for a counterexample critically depends on the efficiency of its backend SAT solver. Modern SAT solvers [4, 12] have shown the effectiveness of adding redundant conflict clauses, which greatly improve the performance by pruning the search space, and by affecting the choice of decision variables. In a similar way, we use the BDD constraints as additional (redundant) constraints on the state variables at each (or some) cycle of the unrolled design, as shown by dark boxes in Figure 2 (a).

For safety properties, we also use the BMC engine to perform a proof by induction with increasing depth  $k$ , with restriction to loop-free paths for completeness [5]. Here, we use the BDDs not as redundant constraints, but as additional constraints to facilitate the proof. This is shown pictorially by the dark boxes labeled  $B^*$  and  $F^*$  in Figure 2 (b). For the base step, after unsatisfiability of the negated property at each cycle up to  $k$  cycles has been checked, we additionally check the satisfiability of the  $B^*$  BDD constraint after  $k$  cycles. If it is unsatisfiable, then the property is proved to be true. However, if  $B^*$  is satisfiable, we proceed with the inductive step, where we use the  $F^*$  BDD to constrain the arbitrary state at the start of the  $k+1$  cycles. Note that this provides an additional *reachability invariant*, which can potentially allow the inductive step to succeed with BMC. It is interesting to note that the backward set  $B^*$

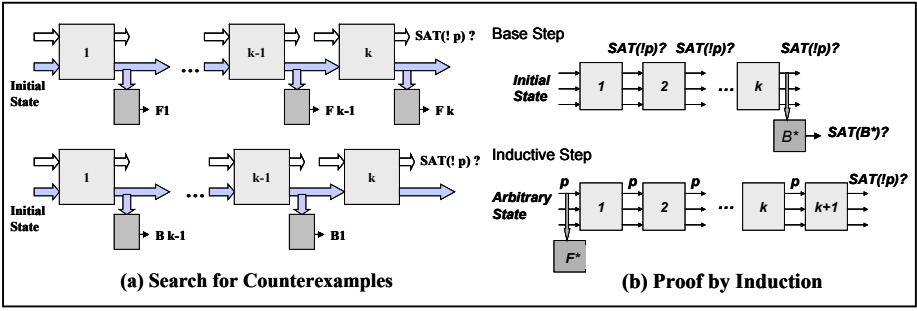


Fig. 2. Use of BDD Constraints in BMC Applications

complements the forward reasoning of the base step, while the forward set  $F^*$  complements the backward reasoning of the inductive step.

We also use BDD constraints (forward/backward iteration BDDs) for representing sets of abstract counterexamples for simple safety properties, for performing counterexample guided abstraction refinement [13]. We are currently investigating their use for representing *witness graphs* of more general properties [14], in order to guide BMC in its search for a real counterexample, or to aid in refinement.

## 4 Experimental Results

We experimented with *DiVer* for checking safety properties on some large industrial designs, with more than 2k flip-flops, and more than 10k gates in the cone of influence of the property. The experiments were conducted on a Dual Intel Xeon 1700 MHz processor, with 4GB memory, running Linux 7.2, with a time limit of 3 hours. We were able to search quite deep in each design (more than 100 cycles), but could not find a counterexample. Similarly, we attempted deep proofs by induction (of depth more than 40), but the induction proofs did not succeed, despite use of about 20 universal constraints (enforced in every state) provided by the designers.

Next, we tried BMC with BDD constraints in *DiVer*. These experiments were conducted on a 440 MHz. Sun workstation, with 1GB memory, running Solaris 5.7. The results are shown in Table 1, where Columns 2 – 5 report the results for BDD-based analysis on the abstract model, while Columns 6 – 9 report the results for a BMC-based proof by induction on the concrete design, with use of the BDD constraints. We obtained the abstract models automatically from the unconstrained designs, by abstracting away latches farther in the dependency closure of the property variables. For these experiments, we performed only the symbolic traversal on the abstract model, since our BDD-based symbolic model checker did not allow checking on universally constrained paths. Columns 2 – 5 report the size of the abstract model (number of flip-flops #FF, number of gates #G), the CPU time taken for traversal, the number of forward iterations, and the final size of the BDD  $F^*$ , respectively. Columns 6 – 9 report the size of the concrete design, the verification status, and the time and memory used by the BMC engine, respectively. Note that due to the small

size of the abstract models, we could keep the resource requirements for BDDs fairly low. The important point is that despite gross approximations in the abstract model analysis, the BDD reachability invariants were strong enough to let the induction proof go through successfully with BMC in each case. Though neither the BDD-based engine, nor the BMC engine, could individually prove these safety properties, their combination allowed the proof to be completed very easily (in less than a minute).

**Table 1.** Experimental Results for Proof by Induction using BDD-based Reachability Invariant

	BDD-based Abstract Model Analysis				Induction Proof with BDD Constraints on Concrete Design			
	#FF / #G	Time(s)	Depth	Size of F*	#FF / #G	Status	Time(s)	Mem(MB)
D1-p1	41 / 462	1.6	7	131	2198 / 14702	TRUE	0.07	2.72
D2-p2	115 / 1005	15.3	12	677	2265 / 16079	TRUE	0.11	2.84
D3-p3	63 / 1001	18.8	18	766	2204 / 16215	TRUE	0.1	2.85

## References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu: Symbolic Model Checking without BDDs. In: Proceedings of TACAS, vol. 1579, LNCS, 1999.
2. R.E. Bryant: Graph-based algorithms for Boolean function manipulation. In IEEE Transactions on Computers, vol. C-35(8), pp. 677-691, 1986.
3. K. L. McMillan: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, 1993.
4. M. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik: Combining Strengths of Circuit-based and CNF-based SAT Algorithms for a High Performance SAT Solver. In Proceedings of Design Automation Conference, 2002.
5. M. Sheeran, S. Singh, and G. Stalmarck: Checking Safety Properties using Induction and a SAT Solver. In Proceedings of Formal Methods in Computer Aided Design, 2000.
6. J. Baumgartner, A. Kuehlmann, and J. Abraham: Property Checking via Structural Analysis. In Proceedings of Computer Aided Verification, 2002.
7. G. Cabodi, S. Nocco, and S. Quer: SAT-based Bounded Model Checking by means of BDD-based Approximate Traversals. In Proceedings of Design And Test Europe (DATE), 2003.
8. E. M. Clarke, O. Grumberg, and D. Peled: Model Checking. MIT Press, 1999.
9. R. P. Kurshan: Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach. Princeton University Press, 1994.
10. A. Gupta, P. Ashar, and S. Malik: Exploiting Retiming in a Guided Simulation Based Validation Methodology. In Proceedings of CHARME, 1999.
11. A. Kuehlmann and J. Baumgartner: Transformation-based Verification using Generalized Retiming. In Proceedings of Computer Aided Verification, 2001.
12. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik: Chaff: Engineering a Efficient SAT Solver. In Proceedings of Design Automation Conference, 2001.
13. P. Chauhan, E. M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang: Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis. In Proceedings of Formal Methods in Computer Aided Design, 2002.
14. A. Gupta, A. E. Casavant, P. Ashar, X. G. Liu, A. Mukaiyama, K. Wakabayashi: Property-Specific Witness Graph Generation for Guided Simulation. In Proceedings of VLSI Design Conference, 2002.

# TLQSolver: A Temporal Logic Query Checker

Marsha Chechik and Arie Gurfinkel

Department of Computer Science, University of Toronto,  
Toronto, ON M5S 3G4, Canada.  
{chechik, arie}@cs.toronto.edu

## 1 Introduction

*Query checking* was proposed by Chan [2] to speed up design understanding by discovering properties not known *a priori*. A *temporal logic query* is an expression containing a symbol  $?_x$ , referred to as the *placeholder*, which may be replaced by any propositional formula<sup>1</sup> to yield a CTL formula, e.g.  $AG?_x$ ,  $AG(?_x \wedge p)$ . A propositional formula  $\psi$  is a *solution* to a query  $\varphi$  in state  $s$  if substituting  $\psi$  for the placeholder in  $\varphi$  is a formula that holds in state  $s$ . A query  $\varphi$  is *positive* [2] if when  $\psi_1$  is a solution to  $\varphi$  and  $\psi_1 \Rightarrow \psi_2$ , then  $\psi_2$  is also a solution. For example, if  $p \wedge q$  is a solution to  $\varphi$ , then so is  $p$ . For positive queries, we seek to compute a set of strongest propositional formulas that make them true. For example, consider evaluating the query  $AG?_x$ , i.e., “what are the invariants of the system”, on a model in Figure 1(a), ignoring the variable  $m$ .  $(p \vee q) \wedge r$  is the strongest invariant: all others, e.g.,  $p \vee q$  or  $r$ , are implied by it. Thus, it is the solution to this query. In turn, if we are interested in finding the strongest property that holds in all states following those in which  $\neg q$  holds, we form the query  $AG(\neg q \Rightarrow AX?_x)$  which, for the model in Figure 1(a), evaluates to  $q \wedge r$ . Chan also showed [2] that a query is positive iff the placeholder appears under an even number of negations.

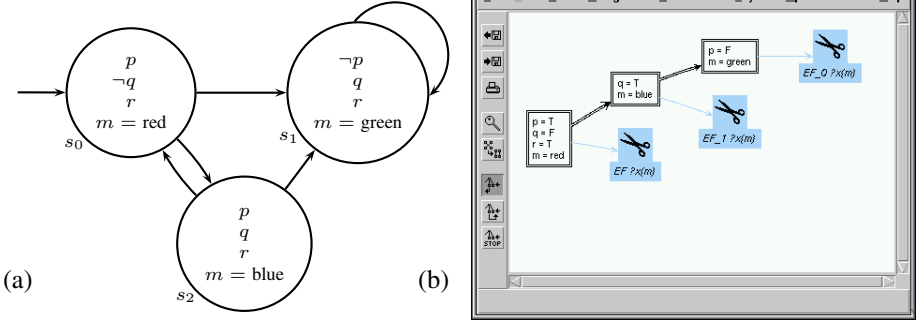
Alternatively, a query is *negative* if a placeholder appears under an odd number of negations. For negative queries, we seek to compute a set of *weakest* propositional formulas that make them true.

In solving queries, we usually want to restrict the atomic propositions that are present in the answer. For example, we may not care about the value of  $r$  and  $m$  in the invariant computed for the model in Figure 1(a). We phrase our question as  $AG(?_x\{p, q\})$ , thus explicitly restricting the propositions of interest to  $p$  and  $q$ . The answer we get is  $p \vee q$ . Given a fixed set of  $n$  atomic propositions of interest, the query checking problem defined above can be solved by taking all  $2^{2^n}$  propositional formulas over this set, substituting them for the placeholder, verifying the resulting temporal logic formulas, tabulating the results and then returning the strongest solution(s) [1]. The number  $n$  of propositions of interest provides a way to control the complexity of query checking in practice, both in terms of computation, and in terms of understanding the resulting answer.

In his paper [2], Chan proposed a number of applications for query checking, mostly aimed at giving more feedback to the user during model checking, by providing a partial explanation when the property holds and diagnostic information when it does not. For example, instead of checking the invariant  $AG(a \vee b)$ , we can evaluate the query

---

<sup>1</sup> A propositional formula is a formula built only from atomic propositions and boolean operators.



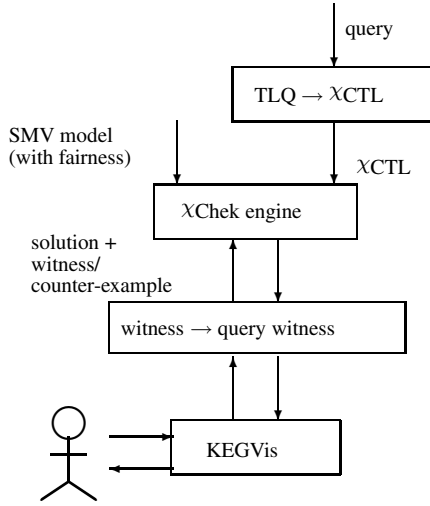
**Fig. 1.** (a) A simple state machine; (b) A witness to query  $EF?_x\{m\}$ .

$AG?_x\{a, b\}$ . Suppose the answer is  $a \wedge b$ , that is,  $AG(a \wedge b)$  holds in the model. We can therefore inform the user of a stronger property and explain that  $a \vee b$  is invariant because  $a \wedge b$  is. We can also use query checking to gather diagnostic information when a CTL formula does not hold. For example, if  $AG(\text{req} \Rightarrow AF \text{ack})$  is *false*, that is, a request is not always followed by an acknowledgment, we can ask *what* can guarantee an acknowledgment:  $AG(?_x \Rightarrow AF \text{ack})$ .

In his work, Chan concentrated on *valid* queries, that is, queries that always have a single strongest solution. All of the queries we mentioned so far are valid. Chan showed that in general it is expensive to determine whether a CTL query is valid. Instead, he identified a syntactic class of CTL queries such that every formula in the class is valid. He also implemented a query-checker for this class of queries on top of the symbolic CTL model-checker SMV.

Queries may also have multiple strongest solutions. Suppose we are interested in exploring successors of the initial state of the model in Figure 1(a), again ignoring variable  $m$ . Forming a query  $EX?_x$ , i.e., “what holds in any of the next states, starting from the initial state  $s_0$ ?”, we get two incomparable solutions:  $p \wedge q \wedge r$  and  $\neg p \wedge q \wedge r$ . Thus, we know that state  $s_0$  has at least two successors, with different values of  $p$  in them. Furthermore, in all of the successors,  $q \wedge r$  holds. Clearly, such queries might be useful for model exploration. Checking queries with multiple solutions can be done using the theoretical framework of Bruns and Godefroid [1]. They extend Chan’s work by showing that the query checking problem with a single placeholder can be solved using an extension of alternating automata.

This paper introduces TLQSolver – a query checker that can decide positive and negative queries with a single or multiple strongest solutions. In addition, it can decide queries with multiple placeholders. TLQSolver is built on top of our existing multi-valued symbolic model-checker  $\chi$ Chek [3]. Our implementation not only allows one to compute solutions to the placeholders, but also gives *witnesses* – paths through the model that explain why solutions are as computed. We also give a few examples of use of TLQSolver, both in domains not requiring witness computation and in those that depend on it. Further uses are described in [6].



**Fig. 2.** Architecture of TLQSolver.

## 2 Implementation and Running Time

The architecture of TLQSolver is given in Figure 2. TLQSolver is implemented on top of our symbolic multi-valued model-checker  $\chi$ Chek [3].  $\chi$ Chek is written in Java, and provides support for both model-checking with fairness and the generation of counter-examples (or witnesses). It uses the *daVinci Presenter* for layout and exploration of counter-examples. TLQSolver takes a query and translates it into  $\chi$ CTL – a temporal logic used by  $\chi$ Chek.  $\chi$ CTL is a multi-valued extension of CTL that includes multi-valued constants and operations on them. Details of this encoding are given in [5]. Output of  $\chi$ Chek is translated into a form expected for query-checking and passed to KEGVis [4] – an interactive counter-example visualization and exploration tool. In addition to visualization, the user can also define a number of strategies for exploration: forward and backward, setting step granularity, choosing witnesses based on size, branching factor, etc.

The running time of solving a query  $\varphi$  on a Kripke structure  $K$  with the state space  $S$  is in  $O(|S| \times |\varphi| \times dd)$ , where  $dd$  is the complexity of computing existential quantification using decision diagrams [5]. The dominating term in  $dd$  is  $|SS(\varphi)|$  – the number of strongest solutions in the computation of  $\varphi$ . For queries about states, e.g.,  $AG(\neg q \Rightarrow AX?_x)$ ,  $|SS(\varphi)|$  is in  $O(|S|)$ , so query checking is in the same time complexity class as model-checking. All queries we used in our applications (see Section 3 and [6]) are in this category. For queries about paths, e.g.,  $EG?_x$ ,  $|SS(\varphi)|$  is in  $O(2^{|S|})$  [7], so the overall running time is in  $O(|S| \times |\varphi| \times 2^{|S|})$ , which is infeasible even for small problems.

To obtain a copy of TLQSolver, please send e-mail to [xchek@cs.toronto.edu](mailto:xchek@cs.toronto.edu).



### 3 Applications

TLQSolver can be effectively used for a variety of model exploration activities. In particular, it can replace several questions to a CTL model-checker to help express reachability properties and discover system invariants and transition guards [6]. For example, it can determine that the query  $AG(\neg q \Rightarrow AX?_x\{q, r\})$  on the model in Figure 1(a) evaluates to  $q \wedge r$ . We now consider a novel application of query-checking – to guided simulation.

The easiest way to explore a model is to simulate its behavior by providing inputs and observing the system behavior through outputs. The user is presented with a list of available next states to choose from. In addition, some model-checkers, such as NuSMV, allow the user to set additional constraints, e.g., to see only those states where  $p$  holds. However, it is almost impossible to use simulation to guide the exploration towards a given objective. Any wrong choice in the inputs in the beginning of the simulation can result in the system evolving into an “uninteresting” behavior. For example, let our objective be the exploration of how the system shown in Figure 1(a) evolves into its different modes (i.e., different values of variable  $m$ ). We have to *guess* which set of inputs results in the system evolving into mode *red* and then which set of inputs yields transition into mode *green*, etc. Thus, the process of exploring the system using a simulation is usually slow and error prone.

In *guided simulation* [6], the user provides a set of higher-level objectives, e.g.  $EF(m=\text{green})$ , and then only needs to choose between the different paths through the system in cases where the objective cannot be met by a single path. Moreover, each choice is given together with the set of objectives it satisfies. Note that this process corresponds closely to that of *planning*, and its version has been realized on top of NuSMV.

Query-checking is a natural framework for implementing guided simulations. The objective is given by a query, and the witness serves as the basis for the simulation. Suppose that the goal is to illustrate how the system given in Figure 1(a) evolves into all of its modes. The following would be the interaction of the user with TLQSolver. The user then expresses the query as  $EF?_x\{m\}$  and uses KEGVis [4] to set his/her preference of the witness with the largest common prefix. The output produced by TLQSolver is shown in Figure 1(b). In this figure, the witness is presented in a proof-like [4] style where state nodes (double lines) are labeled with proof steps that depend on them. The “scissors” symbol on a proof node means that more information is available, e.g. expanding  $EF?_x\{m\}$  tells the user that two more states, indicated by  $EF\_2?_x\{m\}$ , are required to witness the solution to the query. Since our objective was achieved by a single trace, no further user interaction is required.

### References

1. G. Bruns and P. Godefroid. “Temporal Logic Query-Checking”. In *Proceedings of LICS’01*, pages 409–417, Boston, MA, USA, June 2001.
2. William Chan. “Temporal-Logic Queries”. In *Proceedings of CAV’00*, volume 1855 of *LNCS*, pages 450–463, Chicago, IL, USA, July 2000. Springer.
3. M. Chechik, B. Devereux, and A. Gurfinkel. “XChек: A Multi-Valued Model-Checker”. In *Proceedings of CAV’02*, volume 2404 of *LNCS*, July 2002.

4. A. Gurfinkel and M. Chechik. “Proof-like Counter-Examples”. In *Proceedings of TACAS’03*, April 2003. To appear.
5. A. Gurfinkel and M. Chechik. “Temporal Logic Query Checking through Multi-Valued Model Checking”. CSRG Technical Report 457, University of Toronto, Department of Computer Science, January 2003.
6. A. Gurfinkel, B. Devereux, and M. Chechik. “Model Exploration with Temporal Logic Query Checking”. In *Proceedings of FSE’02*, November 2002.
7. S. Hornus and Ph. Schnoebelen. On solving temporal logic queries. In *Proceedings of AMAST’2002*, volume 2422 of *LNCs*, pages 163–177. Springer, 2002.

# Evidence Explorer: A Tool for Exploring Model-Checking Proofs

Yifei Dong, C.R. Ramakrishnan, and Scott A. Smolka

State University of New York, Stony Brook, NY 11794-4400, USA  
{ydong,cram,sas}@cs.sunysb.edu

## 1 Introduction

We present the Evidence Explorer (<http://www.cs.sunysb.edu/~lmc/ee/>), a new tool for assisting users in navigating the proof structure, or *evidence*, produced by a model checker when attempting to verify a system specification for a temporal-logic property. Due to the sheer size of such evidence, single-step traversal is prohibitive and smarter exploration methods are required. The Evidence Explorer enables users to explore evidence through a collection of orthogonal but coordinated *views*. These views allow one to quickly ascertain the overall perception of evidence through consistent visual cues, and easily locate interesting regions by simple drill-down operations. As described in [3], views are definable in *relational graph algebra*, a natural extension of relational algebra to graph structures such as model-checking evidence.

Our experience in using the Evidence Explorer on several case studies of real-life systems indicates that its use can lead to increased productivity due to shortened evidence traversal time. For example, in the case of formally verifying the Sun Microsystems Java meta-locking algorithm for mutual exclusion and freedom from lockout [1], we had to spend nearly an hour to expand and step through one of the generated model-checking proofs using a standard tree browser. With the Evidence Explorer, we not only cut the process to only a couple of minutes but also were able to recognize the key elements instantly and experiment with the specification via more frequent modifications.

## 2 Features and User Interface

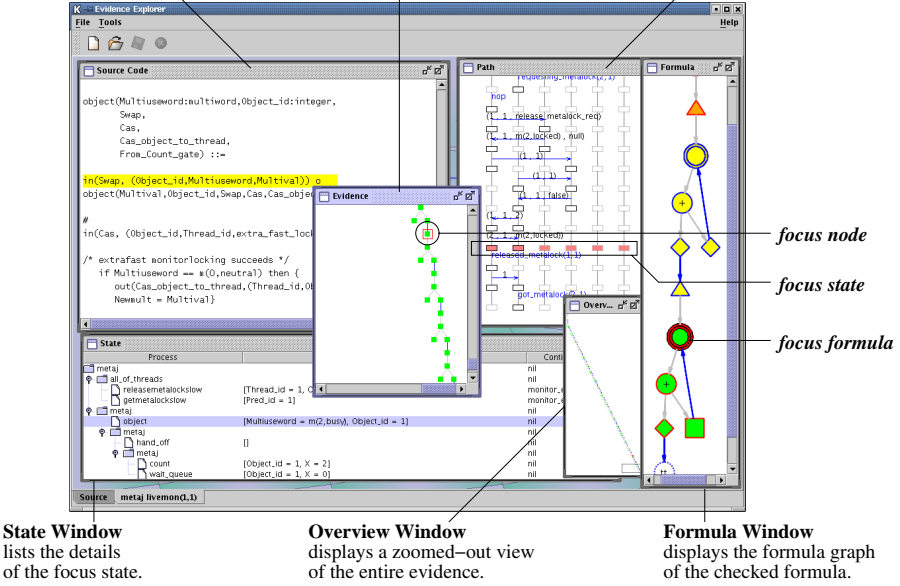
The Evidence Explorer allows users to effectively and intuitively explore the evidence [8,3] produced by a model checker. In this context, an *evidence* is a directed graph whose nodes are pairs of the form  $\langle s, \phi \rangle$ , where  $s$  is a state and  $\phi$  is a (sub-)formula. Such a node represents the assertion that  $s$  satisfies  $\phi$ . Since we use a logic that is closed under negation, the assertion  $s$  *does not* satisfy  $\phi$  is represented by  $\langle s, \neg\phi \rangle$ . An edge in the graph from  $\langle s, \phi \rangle$  to  $\langle s', \phi' \rangle$  means that  $s$  satisfies  $\phi$  only if  $s'$  satisfies  $\phi'$ . A global constraint demands that a least fixed point cannot be the outermost fixed point in a loop, *i.e.* no circular reasoning is allowed for least fixed points. Please see [3] for the full definition of evidence.

The views supported by the Evidence Explorer are organized in six windows as illustrated in Figure 1:

**Source Window**  
highlights lines in source program  
corresponding to the focus state.

**Evidence Window**  
displays the evidence  
or proof structure.

**Path Window**  
displays the MSC capturing the state  
dynamics of the path in the evidence  
from the root to the focus node.



**Fig. 1.** The six synchronized views of the Evidence Explorer.

- The Evidence and Overview windows display the evidence by its spanning tree in zoomed-in and zoom-out resolutions, respectively. Edges not covered by the spanning tree, *i.e.* cross edges and back edges, are indicated using special icons. Each evidence node is painted in the color its formula component assumes in the Formula window.
- The State window lists the details of the focus state (defined below) in a table. A state in a concurrent system may be composed of several substates; each substate may also be a concurrent state having substates. This hierarchy of parallel composition is depicted as a tree. When the user selects a substate of a sequential process from the table, the line of source code corresponding to that process’s control point is highlighted in the Source Code window. The Path window displays a message sequence chart (MSC) capturing the state dynamics of the currently visited path where the focus state belongs.
- The Formula window depicts the formula graph of the checked formula. Several visual cues help users instantly recognize certain properties of the formula and evidence:
  - The *shape* of each formula-graph node is determined by the type of the (sub-)formula’s top-level operator, e.g. box for  $[\cdot]$  operators, circle-plus for disjunctions, and double-circle for fixed points.

- Formula nodes are partitioned into *groups* according to the scope of their fixed points. Nodes in the same group are painted with the same *background color*.
- The *border color* of a formula node indicates the truth value of the evidence nodes associated with the formula: blue, if the truth value is true; red, otherwise.
- When a formula does *not* occur in an evidence, the background of the corresponding node is set to *transparent* so that the user immediately knows that the formula is *vacuous* [2].

Views are synchronized by the *focus node* in the evidence, consisting of the *focus state* and *focus formula*. If the focus is changed in one window, the other windows will automatically update their displays accordingly. Users explore evidence via various ways of changing the focus node:

- Change the focus node directly in the Evidence window. The focus node is highlighted by a red square. The user can move the focus to the parent, children, or siblings of the current focus by pressing arrow keys, or set the new focus to any node by clicking at that node.
- Change the focus state in the Path window. When the user clicks a state in the MSC, the new focus node is set to the node in the evidence path whose state component is the selected state.
- Change the focus formula in the Formula window. When the user clicks on a formula, the new focus is set to an evidence node whose formula component matches the selected formula.

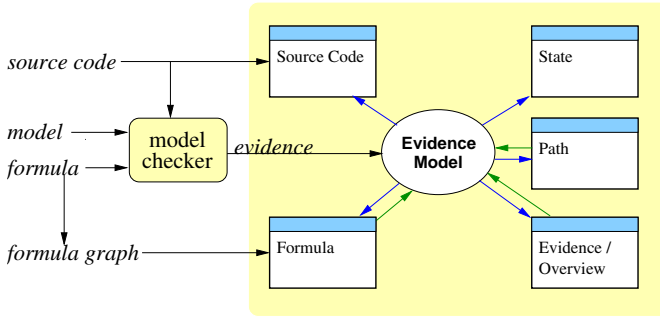
A typical run of the Evidence Explorer starts with the user observing the overall structure of the evidence in the Overview window. He may click the interesting portion to obtain a zoomed-in view in the Evidence window, or explore the evidence by the methods listed above. For debugging purposes, he may also examine the interesting witness/counterexample represented by the MSC in the Path window or key states in the State and Source Code windows.

### 3 Implementation and Extensions

The Evidence Explorer is implemented mainly in Java using the AT&T `dot` package as the graph-layout processor. It uses the XMC verification system [7] as the model-checking back-end. XMC supports the specification of systems in a language based on value-passing CCS and properties in the modal mu-calculus. The architecture of the Evidence Explorer follows the *model-view-controller* (MVC) paradigm [4] as illustrated in Figure 2.

We have designed the Evidence Explorer to be highly extensible and customizable. A planned API will allow tool developers to extend the tool along the following lines:

- utilize alternative graphical components to visualize views;



**Fig. 2.** Architecture of Evidence Explorer.

- define customized views and operations;
- explore multiple evidences;
- plug in other model checkers;
- support compact data structures.

Regarding the last two items, note that we use a conceptual definition of evidence; the physical storage, *e.g.* in a symbolic model checker, can be compressed. When compact data structures are adopted, we map them to the conceptual definition on demand during exploration, without compromising the model checker’s efficiency. With this technique, we can also easily incorporate other forms of evidence such as those in [5,6].

## References

1. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *ECBS 2000*, pages 342–350.
2. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001.
3. Y. Dong, C. Ramakrishnan, and S. A. Smolka. Model checking and evidence exploration. In *ECBS 2003*, pages 214–223.
4. G. Krasner and S. Pope. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *J. Object Oriented Prog.*, 1(3):26–49, 1988.
5. K. S. Namjoshi. Certifying model checkers. In *CAV 2001*, LNCS 2102, pages 2–13.
6. D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *FSTTCS 2001*, LNCS 2245, pages 292–304.
7. C. Ramakrishnan, I. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *CAV 2000*, LNCS 1855, pages 576–580.
8. L. Tan and R. Cleaveland. Evidence-based model checking. In *CAV 2002*, LNCS 2404, pages 455–470.

# HERMES: An Automatic Tool for Verification of Secrecy in Security Protocols<sup>\*</sup>

Liana Bozga, Yassine Lakhnech, and Michaël Périn

VERIMAG, Centre Équation, 2 av. de Vignate, 38610 Gières, France  
{lbozga,lakhnech,perin}@imag.fr

## 1 Introduction

Cryptography is not sufficient for implementing secure exchange of secrets or authentication. Logical flaws in the protocol design may lead to incorrect behavior even under the idealized assumption of perfect cryptography. Most of protocol verification tools are model-checking tools for bounded number of sessions, bounded number of participants and in many case also a bounded size of messages [11,8,5,10]. In general, they are applied to discover flaws in cryptographic protocols. On the contrary, tools based on induction and theorem proving provide a general proof strategy [9,4], but they are either not automatic with exception of [4] or the termination is not guaranteed.

In this paper, we present HERMES, a tool dedicated to the verification of secrecy properties of cryptographic protocols. HERMES places no restriction on the size of messages, neither on the number of participants, nor on the number of sessions. Given a protocol and a secret, HERMES provides either an attack or an invariant on the intruder knowledge that guarantees that the secret will not be revealed by executing the protocol. Moreover, when a protocol is proved correct, it returns a proof tree that can be exploited for certification. HERMES is available online from the authors' webpage.

## 2 The Model and the Verification Method

We give a sketchy idea of the verification method underlying HERMES. A formal and complete presentation of this method can be found in [1]. Cryptographic protocols can be modeled as a set of transitions of the form  $t \rightarrow t'$  where  $t, t'$  are *terms* constructed by applying pairing and the encryption operator  $\{-\}_K$ , to some free variables and the *parameters of a session*, which are the principals, the fresh nonces, and the fresh keys of the session. The intruder is modeled by additional transitions due to Dolev-Yao [6]. They can be seen as a deductive system that describes the messages that the intruder can deduce and forge from the messages sent during the protocols execution. A secrecy goal states that several designated messages (the *secrets*) should not be made public: a secret  $s$  is defined by a term too.

---

<sup>\*</sup> This work and the development of the LAEVA language and accompanying tools EVA TRANS, SECURIFY, CPV and HERMES was supported by the RNTL project EVA (Explication et Vérification Automatique de Protocoles Cryptographiques).

## 2.1 Description of a Protocol and Its Properties

Along the paper we illustrate the main step of the verification on the Needham-Schroeder-Lowe Protocol. Its specification is given below in LAEVA, a high level specification language designed for describing security protocols and their properties. It is compiled by EVATrans into a concrete operational model and a property to check that both constitute the inputs to three automatic verification tools developed in the EVA project: SECURIFY [4], CPV [7] and HERMES [1].

Needham_Schroeder_Lowe	}	<i>pbk</i> and <i>prk</i> are key constructors that take a principal and return an asymmetric key: <i>pbk</i> ( <i>A</i> ) stands for public key of principal <i>A</i> . The private key of <i>A</i> , denoted by <i>prk</i> ( <i>A</i> ), is the inverse of <i>pbk</i> ( <i>A</i> ).
<i>A</i> , <i>B</i> : principal		
<i>Na</i> , <i>Nb</i> : number	}	<i>The knowledge of the principals is needed to generate the operational model of the protocol ; it is used to rule out ambiguities.</i>
keypair <i>pbk</i> , <i>prk</i> (principal)		
everybody knows <i>pbk</i>	}	<i>The protocol specification is close to the standard notation. It describes an ideal session between an initiator (role A) and a responder (role B). The roles A, B, and the nonces Na, Nb that they create are the parameters of a session. With the * symbol, HERMES considers an unbounded number of sessions in parallel. For debugging purpose, it can also run with a fixed number of sessions.</i>
<i>A</i> knows <i>A</i> , <i>B</i> , <i>prk</i> ( <i>A</i> )		
<i>B</i> knows <i>B</i> , <i>prk</i> ( <i>B</i> )	}	<i>secret</i> ( <i>prk</i> ( <i>B@s.A</i> )) means that the private key – of the entity playing the role <i>B</i> , from <i>A</i> 's point of view in session <i>s</i> – is unknown to the intruder. Secrecy hypothesis on keys are needed to reason about encrypted messages.
1. <i>A</i> -> <i>B</i> : { <i>A</i> , <i>Na</i> }_( <i>pbk</i> ( <i>B</i> ))		
2. <i>B</i> -> <i>A</i> : { <i>Na</i> , <i>Nb</i> , <i>B</i> }_( <i>pbk</i> ( <i>A</i> ))	}	<i>HERMES checks that secrecy properties hold *Always and *Globally. The first two claims require that the initial secrets remain secret. The two others asks that the nonces Na and Nb created by role A (resp. role B) in session s are secret.</i>
3. <i>A</i> -> <i>B</i> : { <i>Nb</i> }_( <i>pbk</i> ( <i>B</i> ))		
<i>s.session*</i> { <i>A</i> , <i>B</i> , <i>Na</i> , <i>Nb</i> }	}	
assume		
secret( <i>prk</i> ( <i>A</i> )@ <i>s.A</i> ),	}	
secret( <i>prk</i> ( <i>B</i> )@ <i>s.B</i> ),		
secret( <i>prk</i> ( <i>B@s.A</i> )),	}	
secret( <i>prk</i> ( <i>A@s.B</i> ))		
claim	}	
* <i>A</i> * <i>G</i> secret( <i>prk</i> ( <i>A</i> )@ <i>s.A</i> ),		
* <i>A</i> * <i>G</i> secret( <i>prk</i> ( <i>B</i> )@ <i>s.B</i> ),	}	
* <i>A</i> * <i>G</i> secret( <i>Na@s.A</i> ),		
* <i>A</i> * <i>G</i> secret( <i>Nb@s.B</i> )	}	

This specification leads to three rules parameterized by (*A*, *B*, *N<sub>a</sub>*, *N<sub>b</sub>*) (see below). Then, a session of the protocol is completely defined by instantiating the roles *A* and *B* with some principals, and *N<sub>a</sub>*, *N<sub>b</sub>* with two fresh nonces. On the other hand, *n<sub>1</sub>* and *n<sub>2</sub>* denote free variables which are local to the session.

$$(1.) \frac{-}{\{A, N_a\}_{pbk(B)}}; \quad (2.) \frac{\{A, n_1\}_{pbk(B)}}{\{n_1, N_b, B\}_{pbk(A)}}; \quad (3.) \frac{\{N_a, n_2, B\}_{pbk(A)}}{\{n_2\}_{pbk(B)}};$$

## 2.2 Abstraction and Verification Steps

We reduce the model with unbounded number of sessions in parallel to an finite set of rules which can be applied infinitely and in any order. The rule are



obtained by applying the safe abstraction of [3] that distinguishes only one honest principal  $H$  and the intruder  $I$ . Identifying principals means also identifying their nonces and keys. Hence, the session to consider are reduced to: the session we observe, *i.e.*  $(H, H, N_a, N_b)$ , the sessions with a dishonest participant  $(I, H, N_a^{HH}, N_b^{HH})$  and  $(H, I, N_a^{HI}, N_b^{HI})$ , and the others sessions between honest participants  $(H, H, N_a^{HH}, N_b^{HH})$ , see [1] for details.

Given this abstract model of the protocol, HERMES computes the set of messages that protect the secrets in all sent messages. Intuitively, such protections are terms of the form  $\{x\}_K$  where the inverse key  $K^{-1}$  is not known by the intruder. The hypotheses on secret keys are used to define an initial superset  $P$  of protecting messages, which is then refined by a fixpoint computation. At each step, one transition  $t \rightarrow t'$  of the abstract protocol is analyzed and bad protections are removed from  $P$  or new secrets are added to  $S$ , see [1] for details.

The infinite set  $P$  of protections is represented as two sets  $(G, B)$  of terms. Roughly speaking, the pair  $(G, B)$  is meant for the set of ground messages that are instances of (good) terms in  $G$  but do not unify with (bad) terms of  $B$ .

We provided HERMES with a widening operator that forces termination in the presence of rules that can loop and produce an infinite set of growing bad terms. At termination, HERMES yields an augmented set of secrets  $S'$  ( $S \subseteq S'$ ) and either a representation of a set of protecting messages  $P'$  ( $P' \subseteq P$ ) that protect  $S'$  or an attack in case a secret is revealed.

### 2.3 Application to Our Running Example

We illustrate our approach on the abstract model of the Needham-Schroeder-Lowe Protocol. HERMES starts with the set of secrets  $S = \{N_a, N_b, \text{prk}(H)\}$ , the set of good patterns  $G = \{\{x_s\}_{\text{pbk}(H)}\}$  and an empty set of bad patterns. The fixpoint computation leads to the same set of secrets, the same good patterns and an augmented set of bad protections  $B'$  that consists in four patterns:

$$\{N_a^{HI}, x_s, I\}_{\text{pbk}(H)} ; \{N_a^{HH}, \text{sup}(x_s, I), H\}_{\text{pbk}(H)} ; \{x_s, I\}_{\text{pbk}(H)} ; \{N_a, \text{sup}(x_s, I), H\}_{\text{pbk}(H)}$$

The first one of these bad patterns is obtained from  $\frac{\{N_a^{HI}, x_s, I\}_{\text{pbk}(H)}}{\{x_s\}_{\text{pbk}(I)}}$ , which is the third rule of session  $(H, I, N_a^{HI}, N_b^{HI})$  in the abstract model. Since the conclusion of this rule reveals the secret  $x_s$  to the intruder, we have to remove from the good protections the particular case of messages of the form  $\{N_a^{HI}, x_s, I\}_{\text{pbk}(H)}$ . HERMES stops when no new bad pattern or secret are generated. We then conclude that the protocol satisfies the secrecy property in all initial configurations where the secrets in  $S'$  appears only in messages in compliance with  $(G, B')$ . In our example, the nonces  $N_a$  and  $N_b$  of  $S'$  do not put constraints on the initial configurations since they do not appear at all before the observed session. Finally, the set of secrets  $S'$  restricts the initial configurations to those where the key  $\text{prk}(H)$  was never sent, or sent with a protection in compliance with  $(G, B')$ .

In the original version of the protocol due to Needham-Schroeder, the identity of role B did not appear in message 2. Applied to this version, HERMES requires  $\{H, N_a^{HH}\}_{\text{pbk}(H)}$  to be secret. Obviously, this message can be forged by the intruder. The exploration tree computed by HERMES shows an attack that reveals the secret  $N_b$ . This corresponds to the well known attack discovered by Lowe.

### 3 Experimental Results and Future Work

The following table summarizes the results obtained by HERMES for secrecy properties of protocols from Clark-Jacob's survey [2]. Surprisingly we have not encountered any false attack on any of these protocols, although one could construct a protocol that leads to a false attack. We are currently working on extracting a proof for the Coq prover from the exploration tree provided by HERMES.

Protocol name	result	time (sec)	Protocol name	result	time (sec)
Needham-Schroeder-Lowe	safe	0.02	Yahalom	safe	12.67
Wide Mouthed Frog (modified)	safe	0.01	Kao-Chow	safe	0.07
Neumann-Stubblebine	safe*	0.04	Otway-Rees	safe*	0.02
Andrew Secure RPC	Attack	0.04	Woo and Lam	safe	0.06
Needham-Schroeder Public Key	Attack	0.01	Skeme	safe	0.06
Needham-Schroeder Public Key (with a key server)				Attack	0.90
Needham-Schroeder Symmetric Key				Attack	0.04
Denny Sacco Key Distribution with Public Key				Attack	0.02
ISO Symmetric Key One-Pass Unilateral Authentication				Attack	0.01
ISO Symmetric Key Two-Pass Unilateral Authentication				safe	0.01

\* There is a known attack of the untyped version of the protocol. Discovering this type attack automatically requires to deal with non-atomic keys. This is not yet implemented.

### References

1. L. Bozga, Y. Lakhnech, and M. Périn. Abstract interpretation for secrecy using patterns. In *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2619 of *LNCS*, p. 299–314, 2003.
2. J. Clark and J. Jacob. A survey on authentication protocol literature. Available at the url <http://www.cs.york.ac.uk/~jac/papers/drareviewps.ps>, 1997.
3. H. Comon-Lundh and V. Cortier. Security properties: Two agents are sufficient. In *European Symposium On Programming*, vol. 2618 of *LNCS*, p. 99–113, 2003.
4. V. Cortier, J. Millen, and H. Rueß. Proving secrecy is easy enough. In *Computer Security Foundations Workshop*, p. 97–110, 2001.
5. G. Denker and J. Millen. The CAPSL integrated protocol environment. In *IEEE DARPA Information Survivability Conference and Exposition*, p. 207–222, 2000.
6. D. Dolev and A. C. Yao. On the security of public key protocols. *Transactions on Information Theory*, 29(2):198–208, 1983.
7. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, vol. 1800 of *LNCS*, p. 977–984, 2000.
8. G. Lowe. Casper: A compiler for the analysis of security protocols. In *Computer Security Foundations Workshop*, p. 18–30, 1997.
9. L. Paulson. Proving properties of security protocols by induction. In *Computer Security Foundations Workshop*, p. 70–83, 1997.
10. M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *Computer Security Foundations Workshop*, p. 174–190, 2001.
11. S. Schneider. Verifying authentication protocols with CSP. In *Computer Security Foundations Workshop*, p. 3–17, 1997.

# Iterating Transducers in the Large <sup>\*</sup>

## (Extended Abstract)

Bernard Boigelot, Axel Legay, and Pierre Wolper

Université de Liège,  
Institut Montefiore, B28,  
4000 Liège, Belgium

{boigelot,legay,pw}@montefiore.ulg.ac.be,  
<http://www.montefiore.ulg.ac.be/~{boigelot,legay,pw}/>

**Abstract.** Checking infinite-state systems is frequently done by encoding infinite sets of states as regular languages. Computing such a regular representation of, say, the reachable set of states of a system requires acceleration techniques that can finitely compute the effect of an unbounded number of transitions. Among the acceleration techniques that have been proposed, one finds both specific and generic techniques. Specific techniques exploit the particular type of system being analyzed, e.g. a system manipulating queues or integers, whereas generic techniques only assume that the transition relation is represented by a finite-state transducer, which has to be iterated. In this paper, we investigate the possibility of using generic techniques in cases where only specific techniques have been exploited so far. Finding that existing generic techniques are often not applicable in cases easily handled by specific techniques, we have developed a new approach to iterating transducers. This new approach builds on earlier work, but exploits a number of new conceptual and algorithmic ideas, often induced with the help of experiments, that give it a broad scope, as well as good performance.

## 1 Introduction

If one surveys much of the recent work devoted to the algorithmic verification of infinite-state systems, it quickly appears that regular languages have emerged as a unifying representation formalism for the sets of states of such systems. Indeed, regular languages described by finite automata are a convenient to manipulate, and already quite expressive formalism that can naturally capture infinite sets. Regular sets have been used in the context of infinite sets of states due to unbounded data (e.g. [BG96,FWW97,BW02]) as well as in the context of parametric systems (e.g. [KMM<sup>+</sup>97,PS00]). Of course, whether regular or not, an infinite set of states cannot be computed enumeratively in a finite amount of time. There is thus a need to find techniques for finitely computing the

---

<sup>\*</sup> This work was partially funded by a grant of the “Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées” and by the European IST-FET project ADVANCE (IST-1999-29082).

effect of an unbounded number of transitions. Such techniques can be domain specific or generic. Domain specific results were, for instance, obtained for queues in [BG96,BH97], for integers and reals in [Boi99,BW02], for pushdown system in [FWW97,BEM97], and for lossy channels in [AJ96,ABJ98].

Generic techniques appeared in the context of the verification of parametric systems. The idea used there is that a configuration being a word, a transition relation is a relation on words, or equivalently a language of pairs of words. If this language is regular, it can be represented by a finite state automaton, more specifically a finite-state *transducer*, and the problem then becomes the one of iterating such a transducer. Finite-state transducers are quite powerful (the transition relation of a Turing machine can be modelled by a finite-state transducer), the flip side of the coin being that the iteration of such a transducer is neither always computable, nor regular. Nevertheless, there are a number of practically relevant cases in which the iteration of finite-state transducers can be computed and remains finite-state. Identifying such cases and developing (partial) algorithms for iterating finite-state transducers has been the topic, referred to as “regular model checking”, of a series of recent papers [BJNT00,JN00,Tou01,DLS01,AJNd02].

The question that initiated the work reported in this paper is, whether the generic techniques for iterating transducers could be fruitfully applied in cases in which domain specific techniques had been exclusively used so far. In particular, our goal was to iterate finite-state transducers representing arithmetic relations (see [BW02] for a survey). Beyond mere curiosity, the motivation was to be able to iterate relations that are not in the form required by the domain specific results, for instance disjunctive relations. Initial results were very disappointing: the transducer for an arithmetic relation as simple as  $(x, x + 1)$  could not be iterated by existing generic techniques. However, looking for the roots of this impossibility through a mix of experiments and theoretical work, and taking a pragmatic approach to solving the problems discovered, we were able to develop an approach to iterating transducers that easily handles arithmetic relations, as well as many other cases. Interestingly, it is by using a tool for manipulating automata (LASH [LASH]), looking at examples beyond the reach of manual simulation, and testing various algorithms that the right intuitions, later to be validated by theoretical arguments, were developed. Implementation was thus not an afterthought, but a central part of our research process.

The general approach that has been taken is similar to the one of [Tou01] in the sense that, starting with a transducer  $T$ , we compute powers  $T^i$  of  $T$  and attempt to generalize the sequence of transducers obtained in order to capture its infinite union. This is done by comparing successive powers of  $T$  and attempting to characterize the difference between powers of  $T$  as a set of states and transitions that are added. If this set of added states, or *increment*, is always the same, it can be inserted into a loop in order to capture all powers of  $T$ . However, for arithmetic transducers comparing  $T^i$  with  $T^{i+1}$  did not yield an increment that could be repeated, though comparing  $T^{2^i}$  with  $T^{2^{i+1}}$  did. So, a first idea we used is not to always compare  $T^i$  and  $T^{i+1}$ , but to extract a

sequence of samples from the sequence of powers of the transducer, and work with this sequence of samples. Given the binary encoding used for representing arithmetic relations, sampling at powers of 2 works well in this case, but the sampling approach is general and different sample sequences can be used in other cases. Now, if we only consider sample powers  $T^{i_k}$  of the transducers and compute  $\bigcup_k T^{i_k}$ , this is not necessarily equivalent to computing  $\bigcup_i T^i$ . Fortunately, this problem is easily solved by considering the reflexive transducer, i.e.  $T_0 = T \cup T_I$  where  $T_I$  is the identity transducer, in which case working with an infinite subsequence of samples is sufficient. Finally, for arithmetic transducers, we used the fact that the sequence  $T_0^{2^i}$  can efficiently be computed by successive squaring.

To facilitate the comparison of elements of a sequence of transducers, we work with transducers normalized as reduced deterministic automata. Identifying common parts of successive transducers then amounts to finding isomorphic parts which, given that we are dealing with reduced deterministic automata, can be done efficiently. Working with reduced deterministic automata has advantages, but at the cost of frequently applying expensive determinization procedures. Indeed, during our first experiments, the determinization cost quickly became prohibitive, even though the resulting automata were not excessively large. A closer look showed that this was linked to the fact that the subset construction was manipulating large, but apparently redundant, sets of states. This redundancy was pinpointed to the fact that, in the automata to be determinized, there were frequent inclusion relations between the languages accepted from different states. Formally, there is a partial-order relation on the states of the automaton, a state  $s_1$  being greater than a state  $s_2$  (we say  $s_1$  *dominates*  $s_2$ ) if the language accepted from  $s_1$  includes the language accepted from  $s_2$ . Thus, when applying the subset construction, dominated states can always be eliminated from the sets that are generated. Of course, one needs the dominance relation to apply this but, exploiting the specifics of the context in which determinization is applied, we were able to develop a simple procedure that computes a safe approximation of the dominance relation in time quadratic in the size of the automaton to be determinized.

Once the automata in the sequence being considered are constructed and compared, and that an increment corresponding to the difference between successive elements has been identified, the next step is to allow this increment to be repeated an arbitrary number of times by incorporating it into a loop. There are some technical issues about how to do this, but no major difficulty. Once the resulting “extrapolated” transducer has been obtained, one still needs to check that the applied extrapolation is safe (contains all elements of the sequence) and is precise (contains no more). An easy to check sufficient condition for the extrapolation to be safe is that it remains unchanged when being composed with itself. Checking preciseness is more delicate, but we have developed a procedure that embodies a sufficient criterion for doing so. The idea is to check that any behavior of the transducer with a given number  $k$  of copies of the increment, can be obtained by composing transducers with less than  $k$  copies of the increment.

This is done by augmenting the transducers to be checked with counters and proving that one can restrict these counters to a finite range, hence allowing finite-state techniques to be used.

In our experiments, we were able to iterate a variety of arithmetic transducers. We were also successful on disjunctive relations that could not be handled by earlier specific techniques. Furthermore, to test our technique in other contexts, we successfully applied it to examples of parametric systems and to the analysis of a Petri net.

## 2 Transducers, Arithmetic Transducers and Their Iteration

The underlying problem we are considering is reachability analysis for an infinite-state system characterized by a transition relation  $R$ . Our goal is thus to compute the closure  $R^* = \bigcup_{i \geq 0} R^i$  of  $R$ . In what follows, it will be convenient to also consider the reflexive closure of  $R$ , i.e.  $R \cup I$  where  $I$  is the identity relation, which will be denoted by  $R_0$ ; clearly  $R^* = R_0^*$ .

We will work in the context of regular model checking [BJNT00], in which  $R$  is defined over the set of finite words constructed from an alphabet  $\Sigma$ , is regular and is length preserving (i.e. if  $(w, w') \in R$ , then  $|w| = |w'|$ ). In this case,  $R$  can be defined by a finite automaton over the alphabet  $\Sigma \times \Sigma$ . Such an automaton is called a *transducer* and is defined by a tuple  $T = (Q, \Sigma \times \Sigma, q_0, \delta, F)$  where  $Q$  is the set of states,  $q_0 \in Q$  is the initial state,  $\delta : Q \times (\Sigma \times \Sigma) \rightarrow 2^Q$  is the transition function ( $\delta : Q \times (\Sigma \times \Sigma) \rightarrow Q$  if the automaton is deterministic), and  $F \subseteq Q$  is the set of accepting states.

As it has been shown in earlier work [KMM<sup>+</sup>97, PS00, BJNT00, JN00, Tou01] [DLS01, AJNd02] finite-state transducers can represent the transition relation of parametric systems. Using the encoding of integers by words adopted in [Boi99], finite-state transducers can represent all Presburger arithmetic definable relations plus some base-dependent relations [BHMV94].

If relations  $R_1$  and  $R_2$  are respectively represented by transducers  $T_1 = (Q_1, \Sigma \times \Sigma, q_{01}, \delta_1, F_1)$  and  $T_2 = (Q_2, \Sigma \times \Sigma, q_{02}, \delta_2, F_2)$ , the transducer  $T_{12} = T_2 \circ T_1$  representing the composition  $R_2 \circ R_1$  of  $R_1$  and  $R_2$  is easily computed as  $T_{12} = (Q_1 \times Q_2, \Sigma \times \Sigma, (q_{01}, q_{02}), \delta_{12}, F_1 \times F_2)$ , where  $\delta((q_1, q_2), (a, b)) = \{(q'_1, q'_2) \mid (\exists c \in \Sigma)(q'_1 \in \delta_1(q_1, (a, c)) \text{ and } q'_2 \in \delta_2(q_2, (c, b)))\}$ . Note that even if  $T_1$  and  $T_2$  are deterministic w.r.t.  $\Sigma \times \Sigma$ ,  $T_{12}$  can be nondeterministic.

To compute the closure  $R^*$  of a relation represented by a transducer  $T$ , we need to compute  $\bigcup_{i \geq 0} T^i$ , which is a priori an infinite computation and hence we need a *speed up* technique. In order to develop such a technique, we will consider the reflexive closure  $R_0$  of  $R$  and use the following result.

**Lemma 1.** *If  $R_0$  is a reflexive relation and  $s = s_1, s_2, \dots$  is an infinite subsequence of the natural numbers, then  $\bigcup_{i \geq 0} R_0^i = \bigcup_{k \geq 0} R_0^{s_k}$ .*

The lemma follows directly from the fact that for any  $i \geq 0$ , there is an  $s_k \in s$  such that  $s_k > i$  and that, since  $R_0$  is reflexive,  $(\forall j \leq i)(R_0^j \subseteq R_0^i)$ .

Thus, if we use the transducer  $T_0$  corresponding to the reflexive relation  $R_0$ , it is sufficient to compute  $\bigcup_{k \geq 0} R_0^{s_k}$  for an infinite sequence  $s = s_1, s_2, \dots$  of “sample points”. Note that when the sampling sequence consists of powers of 2, the sequence of transducers  $T_0^{2^k}$  can be efficiently computed by using the fact that  $T_0^{2^{k+1}} = T_0^{2^k} \circ T_0^{2^k}$ .

### 3 Detecting Increments

Consider a reflexive transducer  $T_0$  and a sequence  $s_1, s_2, \dots$  of sampling points. Our goal is to determine whether for each  $i > 0$ , the transducer  $T_0^{s_{i+1}}$  differs from  $T_0^{s_i}$  by some additional constant finite-state structure. One cannot however hope to check explicitly such a property among an infinite number of sampled transducers. Our strategy consists in comparing a finite number of successive transducers until either a suitable increment can be guessed, or the procedure cannot be carried on further.

For each  $i > 0$ , let  $T_0^{s_i} = (Q^{s_i}, \Sigma \times \Sigma, q_0^{s_i}, \delta^{s_i}, F^{s_i})$ . We assume (without loss of generality) that these transducers are deterministic w.r.t.  $\Sigma \times \Sigma$  and minimal. To identify common parts between two successive transducers  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  we first look for states of  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  from which identical languages are accepted. Precisely, we want to construct a relation  $E_f^{s_i} \subseteq Q^{s_i} \times Q^{s_{i+1}}$  such that  $(q, q') \in E_f^{s_i}$  iff the language accepted from  $q$  in  $T_0^{s_i}$  is identical to the language accepted from  $q'$  in  $T_0^{s_{i+1}}$ . Since we are dealing with minimized deterministic transducers, the *forwards equivalence*  $E_f^{s_i}$  is one-to-one (though not total) and can easily be computed by partitioning the states of the joint automaton  $(Q^{s_i} \cup Q^{s_{i+1}}, \Sigma \times \Sigma, q_0^{s_i}, \delta^{s_i} \cup \delta^{s_{i+1}}, F^{s_i} \cup F^{s_{i+1}})$  according to their accepted language. This operation is easily carried out by Hopcroft’s finite-state minimization procedure [Hop71]. Note that because the automata are reduced deterministic, the parts of  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  linked by  $E_f^{s_i}$  are isomorphic, incoming transitions being ignored.

Next, we search for states of  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  that are reachable from the initial state by identical languages. Precisely, we want to construct a relation  $E_b^{s_i} \subseteq Q^{s_i} \times Q^{s_{i+1}}$  such that  $(q, q') \in E_b^{s_i}$  iff the language accepted in  $T_0^{s_i}$  when  $q$  is taken to be the unique accepting state is identical to the language accepted in  $T_0^{s_{i+1}}$  when  $q'$  is taken to be the unique accepting state. Since  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  are deterministic and minimal, the *backwards equivalence*  $E_b^{s_i}$  can be computed by forward propagation, starting from the pair  $(q_0^{s_i}, q_0^{s_{i+1}})$  and exploring the parts of the transition graphs of  $T_0^{s_i}$  and  $T_0^{s_{i+1}}$  that are isomorphic to each other, if transitions leaving these parts are ignored.

Note that taking into account the reduced deterministic nature of the automata we are considering, the relations  $E_f^{s_i}$  and  $E_b^{s_i}$  loosely correspond to the forwards and backwards bisimulations used in [DLS01, AJNd02].

We are now able to define our notion of finite-state “increment” between two successive transducers, in terms of the relations  $E_f^{s_i}$  and  $E_b^{s_i}$ .

**Definition 1.** The transducer  $T_0^{s_{i+1}}$  is incrementally larger than  $T_0^{s_i}$  if the relations  $E_f^{s_i}$  and  $E_b^{s_i}$  cover all the states of  $T_0^{s_i}$ . In other words, for each  $q \in Q^{s_i}$ , there must exist  $q' \in Q^{s_{i+1}}$  such that  $(q, q') \in E_f^{s_i} \cup E_b^{s_i}$ .

**Definition 2.** If  $T_0^{s_{i+1}}$  is incrementally larger than  $T_0^{s_i}$ , then the set  $Q^{s_i}$  can be partitioned into  $\{Q_b^{s_i}, Q_f^{s_i}\}$ , such that

- The set  $Q_f^{s_i}$  contains the states  $q$  covered by  $E_f^{s_i}$ , i.e., for which there exists  $q'$  such that  $(q, q') \in E_f^{s_i}$ ;
- The set  $Q_b^{s_i}$  contains the remaining states<sup>1</sup> of  $Q^{s_i}$ .

The set  $Q^{s_{i+1}}$  can now be partitioned into  $\{Q_H^{s_{i+1}}, Q_{I_0}^{s_{i+1}}, Q_T^{s_{i+1}}\}$ , where

- The head part  $Q_H^{s_{i+1}}$  is the image by  $E_b^{s_i}$  of the set  $Q_b^{s_i}$ ;
- The tail part  $Q_T^{s_{i+1}}$  is the image by  $E_f^{s_i}$  of the set  $Q_f^{s_i}$ , dismissing the states that belong to  $Q_H^{s_{i+1}}$  (our intention is to have an unmodified head part);
- The increment  $Q_{I_0}^{s_{i+1}}$  contains the states that do not belong to either  $Q_H^{s_{i+1}}$  or  $Q_T^{s_{i+1}}$ .

These definitions are illustrated in the first two lines of Figure 1. Note that given the definition used, the transitions between the head part, increment and tail part must necessarily be in the direction shown in the figure.

Our expectation is that when moving from one transducer to the next in the sequence, the increment will always be the same. We formalize this by defining the incremental growth of a sequence of transducers.

**Definition 3.** The sequence of sampled transducers  $T_0^{s_i}, T_0^{s_{i+1}}, \dots, T_0^{s_{i+k}}$  grows incrementally if

- for each  $j \in [0, k-1]$ ,  $T_0^{s_{i+j+1}}$  is incrementally larger than  $T_0^{s_{i+j}}$ ;
- for each  $j \in [1, k-1]$ , the increment  $Q_{I_0}^{s_{i+j+1}}$  is the image by  $E_b^{s_{i+j}}$  of the increment  $Q_{I_0}^{s_{i+j}}$ .

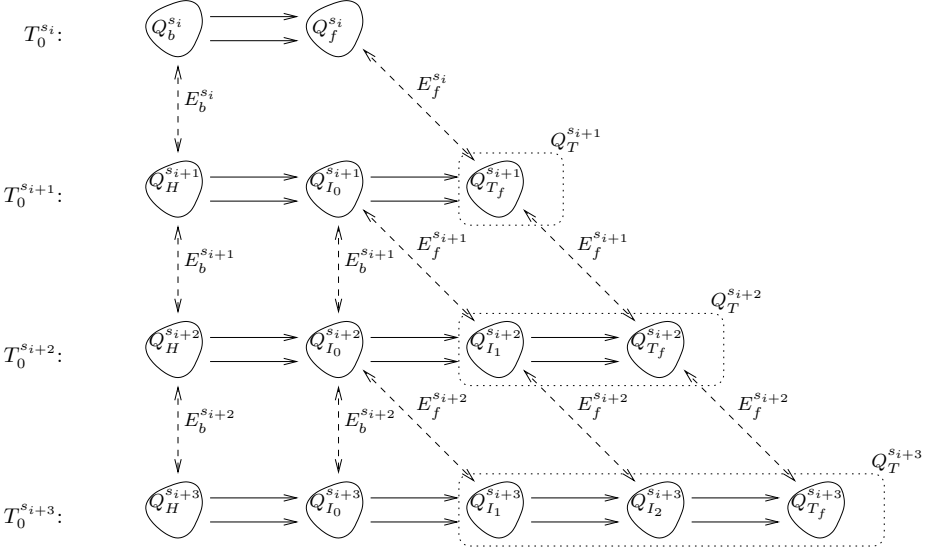
Consider a sequence  $T_0^{s_i}, T_0^{s_{i+1}}, \dots, T_0^{s_{i+k}}$  that grows incrementally. The tail part  $Q_T^{s_{i+j}}$  of  $T_0^{s_{i+j}}$ ,  $j \in [2, \dots, k]$ , will then consist of  $j-1$  copies of the increment plus a part that we will name the *tail-end part*. Precisely,  $Q_T^{s_{i+j}}$  can be partitioned into  $\{Q_{I_1}^{s_{i+j}}, Q_{I_2}^{s_{i+j}}, \dots, Q_{I_{j-1}}^{s_{i+j}}, Q_{T_f}^{s_{i+j}}\}$ , where

- for each  $\ell \in [1, \dots, j-1]$ , the *tail increment*  $Q_{I_\ell}^{s_{i+j}}$  is the image by the relation  $E_f^{s_{i+j-1}} \circ E_f^{s_{i+j-2}} \circ \dots \circ E_f^{s_{i+j-\ell}}$  of the “head” increment  $Q_{I_0}^{s_{i+j-\ell}}$ , where “ $\circ$ ” denotes the composition of relations;
- the *tail-end set*  $Q_{T_f}^{s_{i+j}}$  contains the remaining elements of  $Q_T^{s_{i+j}}$ .

The situation is illustrated in Figure 1.

<sup>1</sup> Definition 1 implies that these states must therefore be covered by  $E_b^{s_i}$ ; the fact that states covered both by  $E_b^{s_i}$  and  $E_f^{s_i}$  are placed in  $Q_f^{s_i}$  is arbitrary, its consequence is that when successive transducers are compared, the part matched to  $Q_f^{s_i}$ , rather than the part matched to  $Q_b^{s_i}$  will grow.





**Fig. 1.** Incrementally-growing sequence of transducers.

Focusing on the last transducer  $T_0^{s_{i+k}}$  in a sequence of incrementally growing transducers, its head increment  $Q_b^{s_{i+k}}$  and all the tail increments  $Q_{I_\ell}^{s_{i+k}}$ ,  $\ell \in [1, k-1]$  appearing in its tail part  $Q_T^{s_{i+k}}$  are images of the increment  $Q_{I_0}^{s_{i+1}}$  by a combination of forwards and backwards equivalences. Indeed, by Definition 3, each tail increment is the image of a previous increment by a composition of forwards equivalences, and each head increment is the image of the previous one by a backwards equivalence. Thus, the transition graphs internal to all increments are isomorphic to that of  $Q_{I_0}^{s_{i+1}}$ , and hence are isomorphic to each other.

Our intention is to extrapolate the transducer  $T_0^{s_{i+k}}$  by adding more increments, following a regular pattern. In order to do this, we need to compare the transitions leaving different increments. We use the following definition.

**Definition 4.** Let  $T_0^{s_{i+k}}$  be the last transducer of an incrementally growing sequence, let  $Q_{I_0}^{s_{i+k}}, \dots, Q_{I_{k-1}}^{s_{i+k}}$  be the isomorphic increments detected within  $T_0^{s_{i+k}}$ , and let  $Q_{T_f}^{s_{i+k}}$  be its “tail end” set. Then, an increment  $Q_{I_\alpha}^{s_{i+k}}$  is said to be communication equivalent to an increment  $Q_{I_\beta}^{s_{i+k}}$  iff, for each pair of corresponding states  $(q, q')$ ,  $q \in Q_{I_\alpha}^{s_{i+k}}$  and  $q' \in Q_{I_\beta}^{s_{i+k}}$ , and  $a \in \Sigma \times \Sigma$ , we have that, either

- $\delta(q, a) \in Q_{I_\alpha}^{s_{i+k}}$  and  $\delta(q', a) \in Q_{I_\beta}^{s_{i+k}}$ , hence leading to corresponding states by the existing isomorphism,
- $\delta(q, a)$  and  $\delta(q', a)$  are both undefined,
- $\delta(q, a)$  and  $\delta(q', a)$  both lead to the same state of the tail end  $Q_{T_f}^{s_{i+k}}$ , or
- there exists some  $\gamma$  such that  $\delta(q, a)$  and  $\delta(q', a)$  lead to corresponding states of respectively  $Q_{I_{\alpha+\gamma}}^{s_{i+k}}$  and  $Q_{I_{\beta+\gamma}}^{s_{i+k}}$ .

In order to extrapolate  $T_0^{s_{i+k}}$ , we simply insert extra increments between the head part of  $T_0^{s_{i+k}}$  and its head increment  $Q_{I_0}^{s_{i+k}}$ , and define the transitions leaving them in order to make these extra increments communication equivalent to  $Q_{I_0}^{s_{i+k}}$ . Of course, before doing so, it is heuristically sound to check that a sufficiently long prefix of the increments of  $T_0^{s_{i+k}}$  are communication equivalent with each other.

## 4 Extrapolating Sequences of Transducers and Correctness

Consider a transducer  $T_{e_0}$  to which extrapolation is going to be applied. The state set of this transducer can be decomposed in a head part  $Q_H$ , a series of  $k$  increments  $Q_{I_0}, \dots, Q_{I_{k-1}}$  and a tail end part  $Q_{T_f}$ . Repeatedly adding extra increments as described at the end of the previous section yields a series of extrapolated transducers  $T_{e_1}, T_{e_2}, \dots$ . Our goal is to build a single transducer that captures the behaviors of the transducers in this sequence, i.e. a transducer  $T_{e_*} = \bigcup_{i \geq 0} T_{e_i}$ . The transducer  $T_{e_*}$  can simply be built from  $T_{e_0}$  by adding transitions according to the following rule.

*For each state  $q \in Q_{I_0} \cup Q_H$  and  $a \in \Sigma \times \Sigma$ , if  $\delta(q, a)$  leads to a state  $q'$  in an increment  $Q_{I_j}$ ,  $1 \leq j \leq k-1$ , then add transitions from  $q$  labelled by  $a$  to the state corresponding to  $q'$  (by the increment isomorphism) in each of the increments  $Q_{I_\ell}$  with  $0 \leq \ell < j$ .*

The added transitions, which include loops (transitions to  $Q_{I_0}$  itself) allow  $T_{e_*}$  to simulate the computations of any of the  $T_{e_i}$ ,  $i \geq 0$ . Conversely, it is fairly easy to see all computations generated using the added transitions correspond to a computation of some  $T_{e_i}$ . Note that the addition of transitions yields a nondeterministic transducer, which needs to be determinized and reduced to be in canonical form.

Having thus constructed an extrapolated transducer  $T_{e_*}$ , it remains to check whether this transducer accurately corresponds to what we really intend to compute, i.e.  $\bigcup_{i \geq 0} T^i$ . This is done by first checking that the extrapolation is *safe*, in the sense that it captures all behaviors of  $\bigcup_{i \geq 0} T^i$ , and then checking that it is *precise*, i.e. that it has no more behaviors than  $\bigcup_{i \geq 0} T^i$ . Both conditions are checked using sufficient conditions.

**Lemma 2.** *The transducer  $T_{e_*}$  is a safe extrapolation if  $L(T_{e_*} \circ T_{e_*}) \subseteq L(T_{e_*})$ .*

Indeed, we have that  $L(T_0) \subseteq L(T_{e_*})$  and thus by induction that  $L(T_0^i) \subseteq L(T_{e_*})$  (recall that  $T_0$  is reflexive).

Determining whether the extrapolation is precise is a more difficult problem. The problem amounts to proving that any word accepted by  $T_{e_*}$ , or equivalently by some  $T_{e_i}$ , is also accepted by an iteration  $T_0^j$  of the transducer  $T_0$ . The idea is to check that this can be proved inductively. The property is true by construction for the transducer  $T_{e_0}$  from which the extrapolation sequence is built. If we can

also prove that, if the property holds for all  $j < i$ , then it also holds for  $i$ , we are done. For this last step, we resort to the following sufficient condition.

**Definition 5.** *A sequence of extrapolated transducers  $T_{e_i}$  is inductively precise if, for all  $i$  and word  $w \in L(T_{e_i})$ , there exist  $j, j' < i$  such that  $w \in L(T_{e_j} \circ T_{e_{j'}})$ .*

To check inductive preciseness, we use automata with counters, the counters being used to count the number of visits to the iterated increment. Three counters are used and we are thus dealing with an undecidable class of automata but, using a usually met “synchronized” condition on the counters, the problem can be reduced to a finite-state one. Details will be given in the full paper.

## 5 Using Dominance to Improve Efficiency

Since to ease the comparison of successive transducers we work with reduced deterministic automata, and since transducer composition usually introduces nondeterminism, each transducer composition step implies a potentially costly determinization procedure. Indeed, our experiments showed that this step could be very resource consuming, even though the resulting transducer was not that much larger than the ones being combined. It thus seemed likely that the transducers we were using had some structure that kept them from growing when being composed. If this structure could be exploited, it would be possible to substantially improve the efficiency of the determinization steps.

Looking at the states generated during these steps, it appeared that they corresponded to large, but vastly redundant, sets of states of the nondeterministic automaton. This redundancy is due to the fact that there are frequent inclusion relations between the languages accepted from different states of the transducer. We formalize this observation with the following notion of *dominance*, similar to the concept used in the ordered automata of [WB00].

**Definition 6.** *Given a nondeterministic finite automaton  $A = (Q, \Sigma, \delta, q_0, F)$ , let  $A_q$  be the automaton  $A = (Q, \Sigma, \delta, q, F)$ , i.e.  $A$  where the initial state is  $q$ . We say that a state  $q_1$  dominates a state  $q_2$  (denoted  $q_1 \geq q_2$ ) if  $L(A_{q_2}) \subseteq L(A_{q_1})$ .*

Clearly, when applying a subset construction, each subset that is generated can be simplified by eliminating dominated states. However, in order to use this, we need to be able to efficiently compute the dominance relation.

A first step is to note that, for deterministic automata, this can be done in quadratic time, by computing the synchronized product of the automaton with itself, and checking reachability conditions on this product. The problem of course is that the automaton to which the determinization and minimization procedure is applied is not deterministic. However, it is obtained from deterministic automata by the composition procedure described in Section 2, and it is possible to approximate the dominance relation of the composed transducer using the dominance relation of the components. Indeed, it is easy to see that if  $q_1 \geq q'_1$  in  $T_1$ , and  $q_2 \geq q'_2$  in  $T_2$ , then  $(q_1, q_2) \geq (q'_1, q'_2)$  in  $T_2 \circ T_1$ ; this being

only sufficient since we can have dominance in the composed transducer without having dominance in the components. Nevertheless, the dominance relation obtained by combining the component relations is a safe approximation and has proven to be quite satisfactory in practice.

## 6 Experiments

The results presented in this paper have been tested on a series of case studies. The prototype implementation that has been used relies in part on the LASH package [LASH] for automata manipulation procedures, but implements the specific algorithms needed for transducer implementation. It is a prototype in the sense that the implementation is not at all optimized, that the interfaces are still rudimentary, that the implementation of the preciseness criterion is not fully operational, and that the increment detection procedure that is implemented is not yet the final one.

As a first series of test cases, we used transducers representing arithmetic relations, such as  $(x, x + k)$  for many values of  $k$ . Turning to examples with multiple variables, the closure of the transducers encoding the relations  $((x, y, z), (z + 1, x + 2, y + 3))$  and  $((w, x, y, z), (w + 1, x + 2, y + 3, z + 4))$  were successfully computed. In addition, we could also handle the transducer encoding the transition relation of a Petri net arithmetically represented by  $((x, y), (x + 2, y - 1)) \cup ((x, y), (x - 1, y + 2)) \cap \mathbb{N}^2 \times \mathbb{N}^2$ . An interesting aspect of this last example is that it is disjunctive and can not be handled by the specific techniques of [Boi99]. In all these examples, the sampling sequence consists of the powers of 2. In Table 1 we give the number of states of some transducers that were iterated, of their closure, and of the largest power of the transducer that was constructed.

**Table 1.** Examples of transducers and their iteration.

Relation	$ T_0 $	$ T_0^* $	Max $ T_0^i $
$(x, x + 1)$	3	3	11
$(x, x + 7)$	7	9	91
$(x, x + 73)$	14	75	933
$((x, y), (x + 2, y - 1)) \cup ((x, y), (x - 1, y + 2)) \cap \mathbb{N}^2 \times \mathbb{N}^2$	19	70	1833
$((x, y), (x + 2, y - 1)) \cup ((x, y), (x - 1, y + 2)) \cup ((x, y), (x + 1, y + 1)) \cap \mathbb{N}^2 \times \mathbb{N}^2$	21	31	635
$((w, x, y, z), (w + 1, x + 2, y + 3, z + 4))$	91	251	2680

We also considered the parametric systems which were used as examples in previous work on transducer iteration. We tried the main examples described in [BJNT00, JN00, Tou01, AJNd02] and our tool was able to handle them. In this case, sampling was not needed in the sense that all powers of the transducer were considered.

## 7 Conclusions and Comparison with Other Work

As a tool for checking infinite-state systems, iterating regular transducers is an appealing technique. Indeed, it is, at least in principle, independent of the type of system being analyzed and is a natural generalization of the iteration of finite-state relations represented by BDDs, which has been quite successful.

Will the iteration of regular transducers also have a large impact on verification applications? The answer to this question is still unknown, but clearly the possibility of scaling up the technique will be a crucial success factor. This is precisely the direction in which this paper intends to make contributions. Indeed, we believe to have scaled up techniques for iterating transducers both qualitatively and quantitatively. From the qualitative point of view, the idea of sampling the sequence of approximations of the iterated transducer, as well as our increment detection and closing technique have enabled us to handle arithmetic transducers that were beyond the reach of earlier methods. Arithmetic relations were also considered in [JN00,BJNT00], but for a simple unary encoding, which limits the expressiveness of regular transducers. From the quantitative point of view, systematically working with reduced deterministic automata and using efficiency improving techniques such as dominance has enabled us to work with quite complex transducers of significant, if not really large size. At least, our implemented tool can find iterations well beyond what can be done by visually inspecting, and manually computing with, automata.

Our work definitely builds on earlier papers that have introduced the basic concepts used in the iteration of regular transducers. For instance, our technique for comparing successive approximations of the iterated transducer can be linked to the reduction techniques used in [JN00,BJNT00,DLS01,AJNd02]. However, we work from the point of view of comparing successive approximations, rather than reducing an infinite-state transducer. This makes our technique similar to the widening technique found in [BJNT00,Tou01], but in a less restrictive setting. Furthermore, we have a novel technique to check that the “widened” transducer corresponds exactly to the iterated transducer. Also, some of the techniques introduced in this paper could be of independent interest. For instance, using dominance to improve the determinization procedure could have applications in other contexts.

Techniques for iterating transducers are still in their infancy and there is room for much further work. The set of transducers we have handled is still limited and there are many other examples to explore and to learn from in order to improve our technique. Our implementation can still be substantially improved, which can also lead to further applications and results. Finally, there are a number of possible extensions, one being to handle automata with infinite words, which would lead the way to applying the iteration of transducers to dense real-time systems.

## Acknowledgement

We thank Marcus Nilsson, Parosh Abdulla, Elad Shahar, and Martin Steffen for answering many email questions on their work.

## References

- [ABJ98] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *Proceedings 10th International Conference on Computer Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318, Vancouver, Canada, 1998. Springer.
- [AJ96] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, June 1996.
- [AJNd02] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d’Orso. Regular model checking made simple and efficient. In *Proceedings 13th International Conference on Concurrency Theory (CONCUR)*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, Brno, Czech Republic, 2002. Springer.
- [BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. Mazurkiewicz and J. Winkowski, editors, *Proceedings of the 8th International conference of Concurrency Theory (CONCUR 97)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150, Warsaw, Poland, July 1997. Springer.
- [BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV’96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12, New-Brunswick, NJ, USA, July 1996. Springer-Verlag.
- [BH97] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO channel systems with nonregular sets of configurations. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceeding of 24th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 560–570, Bologna, Italy, 1997. Springer-Verlag.
- [BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and  $p$ -recognizable sets of integers. *Bulletin of the Belgian Mathematical Society*, 1(2):191–238, March 1994.
- [BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and Tayssir Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV’00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.
- [Boi99] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. Collection des publications de la Faculté des Sciences Appliquées de l’Université de Liège, Liège, Belgium, 1999.
- [BW02] Bernard Boigelot and Pierre Wolper. Representing arithmetic constraints with finite automata: An overview. In *Proc. International Conference on Logic Programming (ICLP)*, volume 2401 of *Lecture Notes in Computer Science*, pages 1–19, Copenhagen, July 2002. Springer-Verlag.
- [DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *Proceedings 13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *Lecture Notes in Computer Science*, pages 286–297, Paris, France, 2001. Springer.

- [FWW97] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. In Faron Moller, editor, *Infinity'97, Second International Workshop on Verification of Infinite State Systems*, volume 9 of *Electronic Notes in Theoretical Computer Science*, Bologna, July 1997. Elsevier Science Publishers.
- [Hop71] J. E. Hopcroft. An  $n \log n$  algorithm for minimizing states in a finite automaton. *Theory of Machines and Computation*, pages 189–196, 1971.
- [JN00] B. Jonsson and M. Nilson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proceeding of the 6th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1875 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2000.
- [KMM<sup>+</sup>97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 424–435. Springer, 1997.
- [LASH] The Liège Automata-based Symbolic Handler (LASH). Available at <http://www.montefiore.ulg.ac.be/~boigelot/research/lash/>.
- [PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2000.
- [Tou01] T. Touili. Regular model checking using widening techniques. In *Proceeding of Workshop on Verification of Parametrized Systems (VEPAS'01)*, volume 50 of *Electronic Notes in Theoretical Computer Science*, 2001.
- [WB95] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 21–32, Glasgow, September 1995. Springer-Verlag.
- [WB00] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

# Algorithmic Improvements in Regular Model Checking <sup>★</sup>

Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso <sup>★★</sup>

Dept. of Information Technology, P.O. Box 337, S-751 05 Uppsala, Sweden  
{parosh,bengt,marcusn,juldor}@it.uu.se

**Abstract.** Regular model checking is a form of symbolic model checking for parameterized and infinite-state systems, whose states can be represented as finite strings of arbitrary length over a finite alphabet, in which regular sets of words are used to represent sets of states. In earlier papers, we have developed methods for computing the transitive closure (or the set of reachable states) of the transition relation, represented by a regular length-preserving transducer. In this paper, we present several improvements of these techniques, which reduce the size of intermediate approximations of the transitive closure: One improvement is to pre-process the transducer by *bi-determinization*, another is to use a more powerful equivalence relation for identifying histories (columns) of states in the transitive closure. We also present a simplified theoretical framework for showing soundness of the optimization, which is based on commuting simulations. The techniques have been implemented, and we report the speedups obtained from the respective optimizations.

## 1 Introduction

*Regular model checking* has been proposed as a uniform paradigm for algorithmic verification of parameterized and infinite-state systems [KMM<sup>+</sup>01, WB98, BJNT00]. In regular model checking, states are represented as finite strings over a finite alphabet, while regular sets of words are used as a symbolic representation of sets of states. Furthermore, regular length-preserving transducers represent transition relations. A generic task in regular model checking is to compute a representation for the set of reachable states, or of the transitive closure of the transition relation. Since we are dealing with an infinite state space, standard iteration-based methods for computing transitive closures (e.g., [BCMD92]) are not guaranteed to terminate.

In previous work [JN00, BJNT00, AJNd02], we have developed methods for computing the transitive closure (or the set of reachable states) of a transducer,

---

<sup>★</sup> This work was supported in part by the European Commission (FET project ADVANCE, contract No IST-1999-29082), and by the the Swedish Research Council (<http://www.vr.se/>)

<sup>★★</sup> This author is supported in part by ARTES, the Swedish network for real-time research (<http://www.artes.uu.se/>).



which are complete for transition relations that satisfy a condition of *bounded local depth* (this can be seen as a non-trivial generalization, to the case of transducers, of the condition of “bounded-reversal” for Turing machines [Iba78]).

These techniques (and those by Dams et al. [DLS01]) input a transducer  $T$ , and attempt to construct a finite representation of the union  $T^* = \cup_{i=0}^{\infty} T^i$  of the finite compositions  $T^i$  of  $T$ . The states of  $T^i$  can be seen as “histories” (which we will call *columns*) of length  $i$  of transducer states. There are infinitely many such columns in  $T^*$ , and the challenge is to find a finite-state automaton which is equivalent to  $T^*$ . In [BJNT00], we presented a technique which directly determinizes  $T^*$  by the subset construction, where subsets are represented as regular sets of columns. In addition, subsets are identified using a pre-defined equivalence relation; this ensures termination if  $T$  has bounded local depth. In [AJNd02], we presented a more light-weight technique, which successively computes the approximations  $T^{\leq n} = \cup_{i=1}^n T^i$  for  $n = 1, 2, \dots$ , while quotienting the set of columns with a certain equivalence relation. For implementation, this technique represented a substantial simplification over the heavy automata-theoretic constructions of [BJNT00], and gave significant performance improvements on most of the examples that we have considered.

In this paper, we present several improvements to the techniques of [AJNd02]. Here, we describe the most important improvements (illustrated in an example in the next section):

- *Bi-determinization*: A key element of our techniques [JN00, BJNT00, AJNd02] is the equivalence relation used to identify columns of  $T^*$ . Ideally, this equivalence should be as large as possible while still respecting the constraint that the quotient of  $T^*$  is equivalent to  $T^*$ . Our equivalences are based on so-called *left-* or *right-copying* states. Roughly, a path in the transducer from an initial to a left-copying state, or from a right-copying to a final state, may only copy symbols. Borrowing intuition from rewriting theory, the copying states of a transducer define the “context” for the actual transformation of the word. The equivalence relation in [AJNd02] is based on ignoring the number of successive repetitions of the same left- or right-copying state in columns, but does not cope with sequences of *distinct* left-copying (or right-copying) states. To overcome this, we now pre-process the transducer by *bi-determinization* before the actual computation of the transitive closure. The effect of bi-determinization is that columns with successive distinct left-copying (or right-copying) states can be safely ignored, since they will never occur in any accepting run of  $T^*$ .
- *Coarser equivalence relations*: In addition to adding the bi-determinization phase, we have increased the power of the equivalence relations in [AJNd02]: whereas before we identified columns with one or more repetitions of a left- or right-copying state, we can now in certain contexts also identify columns with zero or more such repetitions, and identify states with  $k$  alternations between left- and right-copying states with states that have 3 alternations, if  $k > 3$ .

- *Improved theoretical framework*: The equivalences are proven sound by an improved theoretical framework, based on simulations which are also rewrite relations. This framework is a generalization and simplification of the work by Dams et al. [DLS01], who use bisimulations. However, the bisimulation-based framework can not prove that the reduction of alternation (to at most 3) is sound.

We have implemented these optimizations, and tested them on a number of parameterized mutual exclusion protocols. The performance improvement is at least an order of magnitude.

**Related Work** Regular model checking was advocated by Kesten et al. [KMM<sup>+</sup>01], and implemented in the Mona [HJJ<sup>+</sup>96] package. Techniques for accelerating regular model checking have been considered in our earlier work [JN00, BJNT00, AJNd02].

Dams et al. [DLS01] present a related approach, introducing a generic framework for quotienting of the transducer  $T^*$  to find a finite-state representation, and which is not limited to length-preserving transductions. Dams et al. find such an equivalence by a global analysis of the current approximation of the transitive closure. It appears that this calculation is very expensive, and the paper does not report successful experimental application of the techniques to examples of similar complexity as in our work. In this paper, we present a generalization of the equivalences defined by Dams et al.

Touili [Tou01] presents a technique for computing transitive closures of regular transducers based on *widening*, and shows that the method is sufficiently powerful to simulate earlier constructions described in [ABJN99] and [BMT01].

The works in [AJMd02, BT02] extend regular model checking to the context of tree transducers. When applied on words, these techniques correspond to the acceleration and widening techniques described in [BJNT00] and therefore do not cover the optimizations presented in this paper.

**Outline** In the next section, we illustrate the problem and the main ideas through an example. In Section 3, we present a general framework to derive equivalence relations which are language preserving under quotienting. We use this framework in Section 4 to define a new equivalence relation. Finally, in Section 5, we report experimental results of an implementation based on the new techniques.

## 2 An Example

In this section, we will present and illustrate the problem and the techniques through a token passing system with a ring topology.

**Preliminaries** Let  $\Sigma$  be a finite alphabet of symbols. Let  $R$  be a regular relation on  $\Sigma$ , represented by a finite-state *transducer*  $T = \langle Q, q_0, \longrightarrow, F \rangle$  where  $Q$  is the (finite) set of states,  $q_0$  is the initial state,  $\longrightarrow \subseteq Q \times (\Sigma \times \Sigma) \times Q$  is the transition

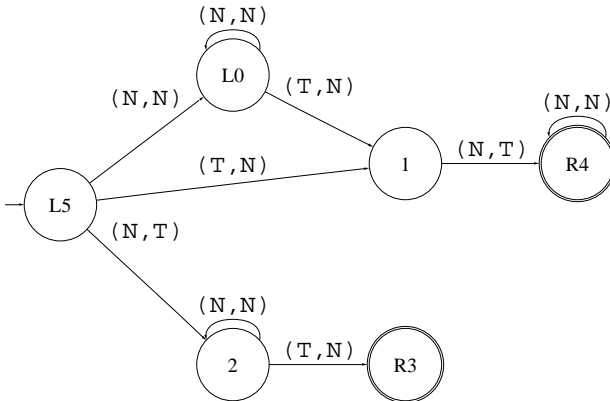
relation, and  $F \subseteq Q$  is the set of accepting states. Note that  $T$  is not required to be deterministic. We use  $q_1 \xrightarrow{(a,b)} q_2$  to denote that  $(q_1, (a, b), q_2) \in \longrightarrow$ . We use a similar infix notation also for the other types of transition relations introduced later in the paper.

The set of *left-copying* states in  $Q$  is the largest subset  $Q_L$  of  $Q$  such that whenever  $q \xrightarrow{(a,a')} q'$  and  $q' \in Q_L$ , then  $a = a'$  and  $q \in Q_L$ . Analogously, the set of *right-copying* states in  $Q$  is the largest subset  $Q_R$  of  $Q$  such that whenever  $q \xrightarrow{(a,a')} q'$  and  $q \in Q_R$ , then  $a = a'$  and  $q' \in Q_R$ . Intuitively, prefixes of left-copying states only copy input symbols to output symbols, and similarly for suffixes of right-copying states. We shall assume that  $Q_L \cap Q_R = \emptyset$  (if not, we can simply decrease  $Q_L$  and  $Q_R$  to satisfy the assumption).

**Example** As an example, we use a token ring passing protocol. In the system, there is an arbitrary number of processes in a ring, where a token is passed in one direction. In our framework, a *configuration* of the system is represented by a finite word  $a_1 a_2 \cdots a_n$  over the alphabet  $\Sigma = \{N, T\}$  where each  $a_i$  represents:

- $T$  - Process number  $i$  has a token.
- $N$  - Process number  $i$  does not have a token.

The transition relation of this system is represented by the transducer below. For example, there is an accepting run  $L5 \xrightarrow{(N,N)} L0 \xrightarrow{(T,N)} 1 \xrightarrow{(N,T)} R4 \xrightarrow{(N,N)}$  accepting the word  $(N, N)(T, N)(N, T)(N, N)$ . This run means that from the configuration  $NTNN$  we can get to the configuration  $NNTN$  in one transition, i.e. process 2 passes the token to process 3.



Note that we label left-copying states by  $Li$  for some  $i$ , and right-copying states by  $Ri$  for some  $i$ .

**Column transducer** Our goal is to construct a transducer that recognizes the relation  $R^*$ , where  $R^* = \cup_{i \geq 0} R^i$ .

Starting from  $T$ , we can in a straight-forward way construct (see also [BJNT00]) a transducer for  $R^*$  whose states, called *columns*, are sequences of states in  $Q$ , where runs of transitions between columns of length  $i$  accept pairs of words in  $R^i$ . More precisely, define the *column transducer* for  $T$  as the tuple  $T^* = \langle Q^*, q_0^*, \Longrightarrow, F^* \rangle$  where

- $Q^*$  is the set of sequences of states of  $T$ ,
- $q_0^*$  is the set of sequences of the initial state of  $T$ ,
- $\Longrightarrow \subseteq (Q^* \times (\Sigma \times \Sigma)) \times Q^*$  is defined as follows: for any columns  $q_1 q_2 \cdots q_m$  and  $r_1 r_2 \cdots r_m$ , and pair  $(a, a')$ , we have  $q_1 q_2 \cdots q_m \xrightarrow{(a, a')} r_1 r_2 \cdots r_m$  iff there are  $a_0, a_1, \dots, a_m$  with  $a = a_0$  and  $a' = a_m$  such that  $q_i \xrightarrow{(a_{i-1}, a_i)} r_i$  for  $1 \leq i \leq m$ ,
- $F^*$  is the set of sequences of accepting states of  $T$ .

It is easy to see that  $T^*$  accepts exactly the relation  $R^*$ : runs of transitions from  $q_0^i$  to columns in  $F^i$  accept transductions in  $R^i$ . The problem is that  $T^*$  has infinitely many states. Our approach is to define an *equivalence*  $\simeq$  between columns of the column transducer and construct the column transducer with equivalent states merged. Under some conditions, the set of equivalence classes we construct is finite. More precisely, given an equivalence relation  $\simeq$ , the *quotient transducer*  $T_{\simeq}$  is defined as  $T_{\simeq} = \langle Q^*/\simeq, \{q_0\}^*/\simeq, \Longrightarrow_{\simeq}, F_{\simeq} \rangle$  where

- $Q^*/\simeq$  is the set of equivalence classes of columns,
- $q_0^*/\simeq$  is the partitioning of  $q_0^*$  with respect to  $\simeq$ , where we assume that  $q_0^*$  is a union of equivalence classes.
- $\Longrightarrow_{\simeq} \subseteq (Q^*/\simeq) \times (\Sigma \times \Sigma) \times (Q^*/\simeq)$  is defined in the natural way as follows. For any columns  $x, x'$  and symbols  $a, a'$ :

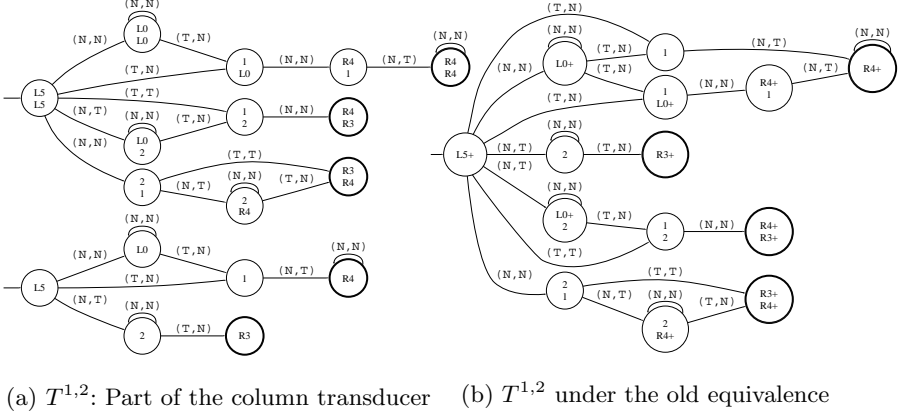
$$x \xrightarrow{(a, a')} x' \quad \Rightarrow \quad [x]_{\simeq} \xrightarrow{(a, a')}_{\simeq} [x']_{\simeq}$$

- $F_{\simeq}$  is the set of equivalence classes in  $Q^*/\simeq$  that have a non-empty intersection with  $F^*$ .

**Example (ctd.)** The part of the column transducer for the token ring system, consisting of states labeled by columns of length one or two, is shown in Figure 1(a). The equivalence  $\simeq$  used in [AJNd02] ignores successive repetitions of the same left-copying and right-copying state, e.g., the column  $R4 R3$  belongs to the equivalence class  $R4^+ R3^+$ , and the column  $1 L0$  belongs to the equivalence class  $1 L0^+$ . The quotient of the transducer in Figure 1(a) under  $\simeq$  is shown in Figure 1(b).

In this paper, we improve on the previous results in the following ways.

**Bi-determinization** In our previous work, the equivalence relation handled the case of repeating the same left-copying or right-copying state, but did not handle sequences of distinct left-copying states or distinct right-copying states. For example, in Figure 1(b), we note that the equivalence classes  $R4^+ R3^+$  and  $R3^+ R4^+$  are disjoint, although they are “equivalent”, and intuitively should be merged.



**Fig. 1.** Part of column transducer under old equivalence

Rather than working out an equivalence relation that identifies suitable sequences of distinct copying states, we solve this problem by preprocessing the transducer by *bi-determinization*. Bi-determinization produces a transducer where the sub-automaton consisting only of the left-copying states is deterministic, and the sub-automaton consisting only of the right-copying states is reverse deterministic. Bi-determinization can be done by a reverse subset construction on the right-copying states. For a bi-deterministic transducer, columns with successive distinct left-copying states are not reachable since two different left-copying states will not have any common prefix. Analogously, columns with successive distinct right-copying states are not productive since two different right-copying states will not have any common suffix. In Figure 2(a), we show the result of bi-determinization of the token ring passing example.

**A new coarser equivalence** In Section 4, we will derive a new equivalence, with coarser equivalence classes. The part of the column transducer for the bi-determinized token ring system, consisting of states labeled by columns of length one or two, optimized with the new equivalence relation, is shown in Figure 2(b). There are two new features in this equivalence:

1. In certain contexts, equivalence classes can now ignore zero or more repetitions of the same copying state. For example, in Figure 2(b) we have the equivalence class  $R5^* 2$ . Using the old equivalence relation (the one in [AJNd02]), we had to represent this equivalence class by the two classes  $2$  and  $R5^+ 2$ .
2. For any left-copying state  $L$  and right-copying state  $R$ , the column  $R L R L R$  will be equivalent to  $R L R$ . In Figure 2(b), there are no such columns present. This becomes important, however, for higher-order approximations of the column transducer.



a pair  $(x, y)$  of columns. The rewrite relation  $\mapsto$  is *generated* by a finite set  $\{(x_1, y_1), \dots, (x_m, y_m)\}$  of rewrite rules if  $\mapsto$  is the least rewrite relation such that  $x_i \mapsto y_i$  for  $i = 1, \dots, m$ . (This means that  $\mapsto$  is the least reflexive and transitive relation such that  $w x_i w' \mapsto w y_i w'$  for any columns  $w, w'$ , and  $i$ ).

We will use the fact that finite simulation relations can be extended to rewrite relations by making them congruences.

**Lemma 1.** *If  $\leq_R$  is a finite forward simulation, then the rewrite relation generated by all pairs in  $\leq_R$  is also a forward simulation. The analogous property holds for backward simulations.*

Using Lemma 1, we can assume that simulations are reflexive and transitive.

Two rewrite relations  $\mapsto_L$  and  $\mapsto_R$  satisfy the *diamond property* if whenever  $x \mapsto_L y$  and  $x \mapsto_R z$ , then there is a  $w$  such that  $y \mapsto_R w$  and  $z \mapsto_L w$ . For rewrite relations that are generated by sets of rewrite rules, the diamond property can be checked efficiently by inspecting all possible critical pairs, i.e., the possible overlaps of left-hand sides of rewrite rules generating  $\mapsto_L$  with left-hand sides of rewrite rules generating  $\mapsto_R$  (cf. [KB70]).

**Lemma 2.** *A pair of rewrite relations  $(\mapsto_L, \mapsto_R)$  satisfying the diamond property induces the equivalence relation  $\simeq$  defined by*

- $x \simeq y$  if and only if there are
- $z$  such that  $x \mapsto_L z$  and  $y \mapsto_R z$ , and
- $z'$  such that  $y \mapsto_L z'$  and  $x \mapsto_R z'$ .

**Lemma 3.** *Let  $\mapsto_L$  and  $\mapsto_R$  be two rewrite relations that satisfy the diamond property. Let  $\simeq$  be the equivalence relation they induce.*

*Then, whenever  $x \simeq y$  and  $x \mapsto_R x'$ , there exists  $y'$  such that  $y \mapsto_R y'$  and  $x' \mapsto_L y'$ .*

*Proof.* To see this, assume that  $x \simeq y$  and  $x \mapsto_R x'$ . This means that there is a  $z$  such that  $x \mapsto_L z$  and  $y \mapsto_R z$ . By the diamond property there is a  $y'$  such that  $x' \mapsto_L y'$  and  $z \mapsto_R y'$ . By the transitivity of  $\mapsto_R$  we infer (from  $y \mapsto_R z$  and  $z \mapsto_R y'$ ) that  $y \mapsto_R y'$ .  $\square$

**Theorem 1.** *Let  $\mapsto_L$  and  $\mapsto_R$  be two rewrite relations that satisfy the diamond property. Let  $\simeq$  be the equivalence relation they induce. If in addition*

- $\mapsto_R$  is a forward simulation,
- $\mapsto_L$  is included in a backward simulation  $\preceq$ ,

*then  $T_{\simeq}$  and  $T^*$  are equivalent.*

*Proof.* Let

$$[x_0]_{\simeq} \xrightarrow{(a_1, b_1)} [x_1]_{\simeq} \xrightarrow{(a_2, b_2)} \dots \xrightarrow{(a_n, b_n)} [x_n]_{\simeq}$$

be a run of  $T_{\simeq}$ . This means that for each  $i$  with  $1 \leq i \leq n$  there are  $x_{i-1}$  and  $y_i$  such that  $x_{i-1} \xrightarrow{(a_i, b_i)} y_i$  and  $x_i \simeq y_i$ . By induction for  $i = 0, 1, 2, \dots, n$ , we find  $x'_i$  and  $y'_i$  such that  $x'_i \xrightarrow{(a_{i+1}, b_{i+1})} y'_{i+1}$  with  $x_i \mapsto_R x'_i$  and  $y'_i \preceq x'_i$ , as follows:

- for  $i = 0$  we take  $x'_0 = y'_0 = x_0$ ,
- for  $i = 1, \dots, n$  we infer from  $x_{i-1} \mapsto_R x'_{i-1}$  that  $y'_i$  exists such that  $x'_{i-1} \xrightarrow{(a_i, b_i)} y'_i$ , and  $y_i \mapsto_R y'_i$ , from which we use Lemma 3 to infer that  $x'_i$  exists with  $x_i \mapsto_R x'_i$  and  $y'_i \mapsto_L x'_i$ ; by inclusion, we get  $y'_i \preceq x'_i$ .

We can now by induction for  $i = n, n-1, \dots, 1$  find  $z_i$  such that  $z_{i-1} \xrightarrow{(a_i, b_i)} z_i$ , and  $y'_i \preceq z_i$ , as follows:

- for  $i = n$  we take  $z_n = x'_n$
- for  $i = n-1, \dots, 0$  we infer from  $y'_{i+1} \preceq z_{i+1}$  and  $x'_i \xrightarrow{(a_{i+1}, b_{i+1})} y'_{i+1}$ , that  $z_i$  exists such that  $z_i \xrightarrow{(a_{i+1}, b_{i+1})} z_{i+1}$ , and  $y'_i \preceq x'_i \preceq z_i$ .

Since  $x_0$  by definition is initial and also  $x_0 = y'_0 \preceq z_0$ , we conclude that  $z_0$  is an initial state. Similarly, because  $x_n$  can be chosen to be accepting (there is one such representative in  $[x_n]_{\simeq}$ ) and  $x_n \mapsto_R x'_n = z_n$ , we conclude that  $z_n$  is accepting.

Therefore, there is a run in  $T^*$  of form

$$z_0 \xrightarrow{(a_1, b_1)} z_1 \xrightarrow{(a_2, b_2)} \dots \xrightarrow{(a_n, b_n)} z_n$$

□

## 4 A Coarse Equivalence

In this section, we derive an equivalence relation using the framework of the previous section.

Let the input transducer  $T$  be the tuple  $\langle Q, q_0, \longrightarrow, F \rangle$ , where  $Q_L \subseteq Q$  is the set of left-copying states,  $Q_R \subseteq Q$  is the set of right-copying states, and  $Q_N = Q \setminus (Q_L \cup Q_R)$  is the set of non-copying states. We shall assume that  $Q_R \cap Q_L = \emptyset$ . For a set  $Q_0$ , let  $Q_0^\epsilon$  denote  $\{\epsilon\} \cup Q_0$ .

Without loss of generality, we can assume that  $T^*$  does not contain any columns with alternations of distinct left- or right-copying states:

**Lemma 4.** *Columns with two consecutive and distinct left- or right-copying states can be removed without changing the language of  $T^*$ .*

*Proof.* A column with two distinct left-copying states is never reachable in  $T^*$  since the left-copying part of  $T$  is deterministic. Similarly, a column with two distinct right-copying states is never productive since the right-copying part is reverse-deterministic. □

We proceed by defining rewrite relations  $\mapsto_R$  and  $\mapsto_L$ .

1. Let  $\mapsto_R$  be the rewrite relation generated by the set of rules of the form  $(q_R, \epsilon)$  and  $(q_R, q_R q_R)$ , where  $q_R$  is a right-copying state.



2. (a) Let  $\preceq$  be the maximal backward simulation on the states of  $T^*$ . This simulation will contain at least the following:
  - $q_L \preceq \epsilon$  for any left-copying state  $q_L$ ;
  - $q_L \preceq q_L q_L$ , for any left-copying state  $q_L$ .
- (b) For columns  $x, y \in Q_L^\epsilon$ , and  $z_1, z_2$ , let  $x \sim_{z_1, z_2} y$  denote that  $z_1 x z_2 \preceq z_1 y z_2$  and  $z_1 y z_2 \preceq z_1 x z_2$ , in other words,  $x$  and  $y$  simulate each other w.r.t.  $\preceq$  in the context of  $z_1$  and  $z_2$ . Note that  $\sim_{z_1, z_2}$  is an equivalence relation on  $Q_L^\epsilon$  for any  $z_1, z_2$ .
- (c) We define  $\mapsto_L$  to be the rewrite relation generated by the rules
  - $(q_L, \epsilon)$  and  $(q_L, q_L q_L)$  for any left-copying state  $q_L$ ,
  - $(z_1 x z_2, z_1 y z_2)$  for any  $x, y \in Q_L^\epsilon$  and  $z_1, z_2 \in Q_N^\epsilon$  such that  $x \sim_{z_1, z_2} y$ .

The following Theorem shows that the rewrite relations induce an equivalence relation with the properties of Theorem 1.

**Theorem 2.** 1.  $\mapsto_R$  is a forward simulation.

2.  $\mapsto_L$  is included in the backward simulation  $\preceq$ .

3.  $\mapsto_L$  and  $\mapsto_R$  have the diamond property.

*Proof.* 1. Follows from the fact that  $q_R$  is right-copying.

2. Follows from the fact that rules of  $\mapsto_L$  are taken from  $\preceq$ .

3. Let  $x \mapsto_L y$  and  $x \mapsto_R z$ . Then  $x = x_1 q_R x_2$  and  $z = x_1 z' x_2$  for  $z' \in \{\epsilon, q_R q_R\}$ . Since the left hand side of each rule of  $\mapsto_L$  does not contain any state from  $Q_R$ , we conclude that  $y = y_1 q_R y_2$  where  $x_1 \mapsto_L y_1$  and  $x_2 = y_2$ , or  $x_2 \mapsto_L y_2$  and  $x_1 = y_1$ . In either case,  $z = x_1 z' x_2 \mapsto_L y_1 z' y_2$ . Furthermore,  $y = y_1 q_R y_2 \mapsto_R y_1 z' y_2$ , completing the diamond.

□

**Corollary 1.** From Theorem 1 and Theorem 2 it follows that the relation  $\simeq$  induced by  $\mapsto_L$  and  $\mapsto_R$  is an equivalence relation such that  $T_\simeq$  and  $T^*$  are equivalent.

**Implementation of the equivalence relation** In the implementation, we have used an approximation of  $\simeq$  which is a finer equivalence relation than  $\simeq$ . Each equivalence class in this approximation is of the form:

$$z_0 e_0 z_1 e_1 z_2 \cdots e_{n-1} z_n$$

where each  $z_i \in Q_N^\epsilon$  and each  $e_i$  is of the form

$$f f_1 \cdots f_m f'$$

where

- $f$  is of the form  $q_L^*, q_L^+, \epsilon$  for some  $q_L \in Q_L$  such that  $f$  is an equivalence class of  $\sim_{z_i, \epsilon}$ .
- $f'$  is of the form  $q_L^*, q_L^+, \epsilon$  for some  $q_L \in Q_L$  such that  $f'$  is an equivalence class of  $\sim_{\epsilon, z_{i+1}}$ .
- If  $m = 0$ , then  $f f'$  is also in an equivalence class of  $\sim_{z_i, z_{i+1}}$ .

- For  $0 < j \leq m$ , the equivalence class  $f_j$  is one of:
  - $q_R^+ q_L^+ (q_R^+ q_L^+)^* q_R^+$  for some  $q_L \in Q_L$ , or
  - $q_L^+$  for some  $q_L \in Q_L$ , or
  - $q_R^+$  for some  $q_R \in Q_R$ .

For example, a typical equivalence class is  $q_0 q_L^* q_R^+ q_1$ , where  $z_0 = q_0$ ,  $z_1 = q_1$  and  $e_0 = q_L^* q_R^+$ . In this case  $q_L \sim_{q_0, \epsilon} \epsilon$  which means that  $q_0 q_L$  simulates  $q_0$  backward. In [AJNd02], we used an equivalence relation with equivalence classes  $q_L^+$  for left-copying states  $q_L$ , and  $q_R^+$  for right-copying states  $q_R$ .

A justification of these equivalence classes can be found in [AJNd03].

## 5 Implementation

We have implemented the equivalence defined in Section 4. We have run several examples to measure the effectiveness of the method, comparing the number of states built. At the end of this section, we mention some more technical improvements.

In our previous work [AJNd02] we have for each algorithm computed the transitive closures for individual *actions* representing one class of statements in the program. Here, we compute the transitive closure of the transducer representing the *entire program*.

We have compared the number of states generated under the following conditions:

- **Old equivalence** This is the old equivalence used in [AJNd02].
- **Bi-determinization** Using the old equivalence but on a bi-deterministic transducer.
- **New equivalence** Using the new equivalence on a bi-deterministic transducer.

The results can be found in Table 1. The computation time for Bakery is about two minutes with the new techniques, implying a tenfold performance improvement. To reduce the states of the transducer for Szymanski, it was intersected with an invariant while Bakery was not, hence the huge difference in the number of states. The invariant was computed using standard reachability analysis augmented with transitive closures of individual statements. Note that this can not be done in general. For example, computing the invariant this way does not work for Bakery.

*Dead by label* In the algorithm, we might generate some states that can be declared dead by only looking at the label of the state. For a bi-deterministic transducer, any label of the form  $L1 L2$  and  $R1 R2$  can be declared dead, see Lemma 4.

*Caching* Many transitions are merged, and many transitions have the same edge. Thus, caching the result of composing edges provides a substantial runtime improvement.

**Table 1.** Number of live (and total) number of states generated

Transducer / Method	Old equivalence	Bi-determinization	New equivalence
Token passing	6 (15)	6 (15)	4 (10)
Token ring passing	68 (246)	58 (230)	25 (86)
Bakery	1793 (5719)	605(1332)	335(813)
Szymanski	20 (47)	16(30)	11(22)

## References

- [ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc. 11<sup>th</sup> Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.
- [AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d’Orso. Regular tree model checking. In *Proc. 14<sup>th</sup> Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.
- [AJNd02] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d’Orso. Regular model checking made simple and efficient. In *Proc. CONCUR 2002, 13<sup>th</sup> Int. Conf. on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130, 2002.
- [AJNd03] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d’Orso. Algorithmic improvements in regular model checking. Technical Report 2003-024, Department of Information Technology, Uppsala University, April 2003.
- [BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking:  $10^{20}$  states and beyond. *Information and Computation*, 98:142–170, 1992.
- [BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc. 12<sup>th</sup> Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.
- [BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *Proc. LICS’ 01 17<sup>th</sup> IEEE Int. Symp. on Logic in Computer Science*. IEEE, 2001.
- [BT02] Ahmed Bouajjani and Tayssir Touili. Extrapolating Tree Transformations. In *Proc. 14<sup>th</sup> Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.
- [DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.
- [HJJ<sup>+</sup>96] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS ’95, 1<sup>th</sup> Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, 1996.
- [Iba78] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25:116–133, 1978.

- [JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00, 6<sup>th</sup> Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.
- [KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon press, 1970.
- [KMM<sup>+</sup>01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.
- [Tou01] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.
- [WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

# Efficient Image Computation in Infinite State Model Checking <sup>★</sup>

Constantinos Bartzis and Tevfik Bultan

Department of Computer Science,  
University of California Santa Barbara  
{bar,bultan}@cs.ucsb.edu

**Abstract.** In this paper we present algorithms for efficient image computation for systems represented as arithmetic constraints. We use automata as a symbolic representation for such systems. We show that, for a common class of systems, given a set of states and a transition, the time required for image computation is bounded by the product of the sizes of the automata encoding the input set and the transition. We also show that the size of the result has the same bound. We obtain these results using a linear time projection operation for automata encoding linear arithmetic constraints. We also experimentally show the benefits of using these algorithms by comparing our implementation with LASH and BRAIN.

## 1 Introduction

Symbolic representations enable verification of systems with large state spaces which cannot be analyzed using enumerative approaches [15]. Symbolic model checking has been applied to verification of infinite-state systems using symbolic representations that can encode infinite sets [13,8,10]. One class of infinite-state systems is systems that can be specified using linear arithmetic formulas on unbounded integer variables. Verification of such systems has many interesting applications such as monitor specifications [20], mutual exclusion protocols [8,10], parameterized cache coherence protocols [9], and static analysis of access errors in dynamically allocated memory locations (buffer overflows) [11].

There are two basic approaches to symbolic representation of linear arithmetic constraints in verification: 1) *Polyhedral representation*: In this approach linear arithmetic formulas are represented in a disjunctive form where each disjunct corresponds to a convex polyhedron. Each polyhedron corresponds to a conjunction of linear constraints [12,13,10]. This approach can be extended to full Presburger arithmetic by including divisibility constraints (which can be represented as equality constraints with an existentially quantified variable) [8,16]. 2) *Automata representation*: An arithmetic constraint on  $v$  integer variables can be represented by a  $v$ -track automaton that accepts a string if it corresponds

---

<sup>★</sup> This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

to a  $v$ -dimensional integer vector (in binary representation) that satisfies the corresponding arithmetic constraint [5,18,19]. For both of these symbolic representations one can implement algorithms for intersection, union, complement, existential quantifier elimination operations, and subsumption, emptiness and equivalence tests, and therefore use them in model checking.

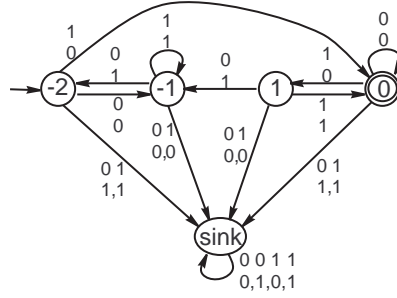
In [17] a third representation was introduced: *Hilbert's basis*. A conjunction of atomic linear constraints  $C$  can be represented as a unique pair of sets of vectors  $(N, H)$ , such that every solution to  $C$  can be represented as the sum of a vector in  $N$  and a linear combination of vectors in  $H$ . Efficient algorithms for backward image computation, satisfiability checking and entailment checking on this representation are discussed in [17]. Based on these results an invariant checker called BRAIN which uses backward reachability is implemented [17]. The experimental results in [17] show that BRAIN outperforms polyhedral representation significantly.

In automata based symbolic model checking, the most time consuming operation is the image computation (either forward or backward). This is due to the fact that image computation involves determinization of automata, an operation with exponential worst case complexity. In this paper we propose new techniques for image computation that are provably efficient for a restricted but quite common class of transition systems. We investigate systems where the transition relation can be characterized as a disjunction of guarded updates of the form *guard*  $\wedge$  *update*, where *guard* is a predicate on current state variables and *update* is a formula on current and next state variables. We assume that the update formula is a conjunction of equality constraints. We show that for almost all such update formulas, image computation can be performed in time proportional to the size of the automata encoding the input set times the size of the automata encoding the guarded update. The size of the result of the image computation has the same bound. We discuss efficient implementation of the algorithms presented in this paper using BDD encoding of automata. Furthermore, we present experimental results that demonstrate the usefulness of our approach and its advantages over methods using other representations.

The rest of the paper is organized as follows. Section 2 gives an overview of automata encoding of Presburger formulas. Section 3 presents our main results. We first define four categories of updates. We present bounds for pre and post-condition computations for each category and we give algorithms that meet the given bounds. In Section 4 we describe the implementation of the given algorithms and discuss how we can integrate boolean variables to our representation. In Section 5 we present the experimental results and in Section 6 we give our conclusions.

## 2 Finite Automata Representation for Presburger Formulas

The representation of Presburger formulas by finite automata has been studied in [5,19,3,2]. Here we briefly describe finite automata that accept the set of natural



**Fig. 1.** An automaton for  $2x - 3y = 2$

number tuples that satisfy a Presburger arithmetic formula on  $v$  variables. All the results we discuss in this paper are also valid for integers. We use natural numbers to simplify the presentation. Our implementation also handles integers.

We encode numbers using their binary representation. A  $v$ -tuple of natural numbers  $(n_1, n_2, \dots, n_v)$  is encoded as a word over the alphabet  $\{0, 1\}^v$ , where the  $i_{th}$  letter in the word is  $(b_{i1}, b_{i2}, \dots, b_{iv})$  and  $b_{ij}$  is the  $i_{th}$  least significant bit of number  $n_j$ . Given a Presburger formula  $\phi$ , we construct a finite automaton  $FA(\phi) = (K, \Sigma, \delta, e, F)$  that accepts the language  $L(\phi)$  over the alphabet  $\Sigma = \{0, 1\}^v$ , which contains all the encodings of the natural number tuples that satisfy the formula.  $K$  is the set of automaton states,  $\Sigma$  is the input alphabet,  $\delta : K \times \Sigma \rightarrow K$  is the transition function,  $e \in K$  is the initial state, and  $F \subseteq K$  is the set of final or accepting states.

For equalities,  $FA(\sum_{i=1}^v a_i \cdot x_i = c) = (K, \Sigma, \delta, e, F)$ , where  $K = \{k \mid \sum_{a_i < 0} a_i \leq k \leq \sum_{a_i > 0} a_i \vee 0 \leq k \leq -c \vee -c \leq k \leq 0\} \cup \{sink\}$ ,  $\Sigma = \{0, 1\}^v$ ,  $e = -c$ ,  $F = \{0\}$ , and the transition function  $\delta$  is defined as:

$$\delta(k, (b_1, \dots, b_v)) = \begin{cases} (k + \sum_{i=1}^v a_i \cdot b_i) / 2 & \text{if } k + \sum_{i=1}^v a_i \cdot b_i \text{ is even, } k \neq sink \\ sink & \text{otherwise} \end{cases}$$

For inequalities,  $FA(\sum_{i=1}^v a_i \cdot x_i < 0) = (K, \Sigma, \delta, e, F)$ , where  $K = \{k \mid \sum_{a_i < 0} a_i \leq k \leq \sum_{a_i > 0} a_i \vee 0 \leq k \leq -c \vee -c \leq k \leq 0\}$ ,  $\Sigma = \{0, 1\}^v$ ,  $e = -c$ ,  $F = \{k \mid k \in K \wedge k < 0\}$ , and the transition function is  $\delta(k, (b_1, \dots, b_v)) = \lfloor (k + \sum_{i=1}^v a_i \cdot b_i) / 2 \rfloor$ . An example automaton for the equation  $2x - 3y = 2$  is shown in Figure 1.

Conjunction, disjunction and negation of constraints can be implemented by automata intersection, union and complementation, respectively. Finally, if some variable is existentially quantified, we can compute a non-deterministic FA accepting the projection of the initial FA on the remaining variables and then determinize it. The size of the resulting deterministic FA can be exponential on the size of the initial FA, i.e.  $|FA(\exists x_i. \phi)| = 2^{|FA(\phi)|}$  in the worst case. In this

paper we show that for many interesting cases we can avoid this exponential blowup. The resulting FA may not accept *all* satisfying encodings (with any number of leading zeros). We can overcome this by recursively identifying all rejecting states  $k$  such that  $\delta(k, (0, 0, \dots, 0)) \in F$ , and make them accepting. Universal quantification can be similarly implemented by the use of the FA complementation.

### 3 Pre- and Post-condition Computations

Two fundamental operations in symbolic verification algorithms are computing the pre- or post-conditions of a set of states (configurations) of a system. One interesting issue is investigating the sizes of the FA that would be generated by the pre- and post-condition operations.

Given a set of states  $S \subseteq \mathbb{Z}^v$  of a system as a relation on  $v$  integer state variables  $x_1, \dots, x_v$  and the transition relation  $R \subseteq \mathbb{Z}^{2v}$  of the system as a relation on the current state and next state variables  $x_1, \dots, x_v, x'_1, \dots, x'_v$ , we would like to compute the pre- and post-condition of  $S$  with respect to  $R$ , where  $pre(S, R) \subseteq \mathbb{Z}^v$  and  $post(S, R) \subseteq \mathbb{Z}^v$ . We consider systems where  $S$  and  $R$  can be represented as Presburger arithmetic formulas, i.e.,  $S = \{(x_1, \dots, x_v) \mid \phi_S(x_1, \dots, x_v)\}$  and  $R = \{(x_1, \dots, x_v, x'_1, \dots, x'_v) \mid \phi_R(x_1, \dots, x_v, x'_1, \dots, x'_v)\}$ , where  $\phi_S$  and  $\phi_R$  are Presburger arithmetic formulas. For example, consider a system with three integer variables  $x_1, x_2$  and  $x_3$ . Let the current set of states be  $S = \{(x_1, x_2, x_3) \mid x_1 + x_2 = x_3\}$  and the transition relation be  $R = \{(x_1, x_2, x_3, x'_1, x'_2, x'_3) \mid x_1 > 0 \wedge x'_1 = x_1 - 1 \wedge x'_2 = x_2 \wedge x'_3 = x_3\}$ . Then the post-condition of  $S$  with respect to  $R$  is  $post(S, R) = \{(x_1, x_2, x_3) \mid x_1 > -1 \wedge x_1 + x_2 = x_3 - 1\}$ . The pre-condition of  $S$  with respect to  $R$  is  $pre(S, R) = \{(x_1, x_2, x_3) \mid x_1 > 0 \wedge x_1 + x_2 = x_3 + 1\}$ .

One can compute  $post(S, R)$  by first conjoining  $\phi_S$  and  $\phi_R$ , then existentially eliminating the current state variables, and finally renaming the variables, i.e.,  $\phi_{post(S, R)}$  is equivalent to  $(\exists x_1 \dots \exists x_v. (\phi_S \wedge \phi_R))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]}$  where  $\psi_{[y \leftarrow z]}$  is the formula generated by substituting  $z$  for  $y$  in  $\psi$ . On the other hand,  $pre(S, R)$  can be computed by first renaming the variables in  $\phi_S$ , then conjoining with  $\phi_R$ , and finally existentially eliminating the next state variables, i.e.,  $\phi_{pre(S, R)}$  is equivalent to  $\exists x'_1 \dots \exists x'_v. (\phi_{S[x_1 \leftarrow x'_1, \dots, x_v \leftarrow x'_v]} \wedge \phi_R)$ . Hence, to compute  $post(S, R)$  and  $pre(S, R)$  we need three operations: conjunction, existential variable elimination and renaming.

As stated earlier, given  $FA(\phi)$  representing the set of solutions of  $\phi$ ,  $FA(\exists x_1, \dots, \exists x_n. \phi)$  can be computed using projection and the size of the resulting FA is at most  $O(2^{|\text{FA}(\phi)|})$ . Note that existential quantification of more than one variable does not increase the worst case complexity since the determinization can be done once at the end, after all the projections are done. As discussed earlier, conjunction operation can be computed by generating the product automaton, and the renaming operation can be implemented as a linear time transformation of the transition function. Hence, given formulas  $\phi_S$  and  $\phi_R$  representing  $S$  and  $R$ , and corresponding FA,  $FA(\phi_S)$  and  $FA(\phi_R)$ , the size of  $FA(\phi_{post(S, R)})$  and  $FA(\phi_{pre(S, R)})$  is  $O(2^{|\text{FA}(\phi_S)| \cdot |\text{FA}(\phi_R)|})$  in the worst case. Be-



low, we show that under some realistic assumptions, the size of the automaton resulting from pre- or post-condition computations is much better.

We assume that the formula  $\phi_R$  defining the transition relation  $R$  is a guarded-update of the form  $\text{guard}(R) \wedge \text{update}(R)$ , where  $\text{guard}(R)$  is a Presburger formula on current state variables  $x_1, \dots, x_v$  and  $\text{update}(R)$  is of the form  $x'_i = f(x_1, \dots, x_v) \wedge \bigwedge_{j \neq i} x'_j = x_j$  for some  $1 \leq i \leq v$ , where  $f : Z^v \rightarrow Z$  is

a linear function. This is a realistic assumption, since in asynchronous concurrent systems, the transition relation is usually defined as a disjunction of such guarded-updates. Also, note that, the post-condition of a transition relation which is a disjunction of guarded-updates is the union of the post-conditions of individual guarded-updates, and can be computed by computing post-condition of one guarded-update at a time. The same holds for pre-condition.

We consider four categories of updates:

1.  $x'_i = c$
2.  $x'_i = x_i + c$
3.  $x'_i = \sum_{j=1}^v a_j \cdot x_j + c$ , where  $a_i$  is odd
4.  $x'_i = \sum_{j=1}^v a_j \cdot x_j + c$ , where  $a_i$  is even

Note that categories 1 and 2 are subcases of categories 4 and 3 respectively. We can prove that pre-condition computation can be performed efficiently for categories 1-4 and post-condition computation can be performed efficiently for categories 2-3. For each of these cases we give algorithms for computing pre- and post-conditions, and derive upper bounds for the time complexity of the algorithms and the size of the resulting automata. We define  $\phi'_S$  as  $\phi_S[x_1 \leftarrow x'_1, \dots, x_v \leftarrow x'_v]$ . The following Theorem will be used later for the complexity proofs.

**Theorem 1.** *Given a formula  $\psi$  of the form  $\phi(x_1, \dots, x_v) \wedge \sum_{j=1}^v a_j \cdot x_j = c$ , where  $a_i$  is odd for some  $0 \leq i \leq v$ , the deterministic automaton  $FA(\exists x_i. \psi)$  can be computed from  $FA(\psi)$  in linear time and it will have the same number of states.*

*Proof.* For all  $(v-1)$ -bit tuples  $\sigma \in \{0, 1\}^{v-1}$  we define  $\sigma_{b_i=b}$  to be the  $v$ -bit tuple resulting from  $\sigma$  if we insert the bit  $b$  in the  $i$ th position of  $\sigma$ . For example if  $\sigma = (1, 1, 0, 0)$  then  $\sigma_{b_3=0} = (1, 1, 0, 0, 0)$  and  $\sigma_{b_3=1} = (1, 1, 1, 0, 0)$ . Let  $FA(\psi) = (K, \{0, 1\}^v, \delta, e, F)$ . Then the non-deterministic automaton  $FA(\exists x_i. \psi)$  is  $(K, \{0, 1\}^{v-1}, \delta', e, F)$ , where  $\delta'(k, \sigma) = \{\delta(k, \sigma_{b_i=0}), \delta(k, \sigma_{b_i=1})\}$ . Since  $a_i$  is odd, we know that  $\delta(k, \sigma_{b_i=0})$  or  $\delta(k, \sigma_{b_i=1})$  is *sink*. This is because  $\forall k \in K, (b_1, \dots, b_v) \in \{0, 1\}^v$  either  $k + \sum_{j \neq i} a_j \cdot b_j$  or  $k + \sum_{j \neq i} a_j \cdot b_j + a_i$  is odd. By the definition of automata for equalities in Section 2, one of the two transitions goes to *sink* state. We also know that transitions that go to the *sink* state in non-deterministic automata can be safely removed, since they can never be part of an accepting path. So in order to determinize  $FA(\exists x_i. \psi)$  we only need to remove from  $\delta'(k, \sigma)$  one of its two members that is *sink*. Figure 2 shows the algorithm that computes deterministic  $FA(\exists x_i. \psi)$  from  $FA(\psi)$ . Clearly, the complexity of the algorithm is  $O(|FA(\psi)|)$ .

Input  $\text{FA}(\psi) = (K, \{0, 1\}^v, \delta, e, F)$ , where  $\psi = \phi \wedge \sum_{j=1}^v a_j \cdot x_j = c$   
integer  $i, 0 \leq i \leq v$   
Output  $\text{FA}(\exists x_i. \psi) = (K, \{0, 1\}^{v-1}, \delta', e, F)$

FOR ALL  $k \in K, \sigma \in \{0, 1\}^{v-1}$  DO  
  IF  $\delta(k, \sigma_{b_i=0}) = \text{sink}$  THEN  
     $\delta'(k, \sigma) = \delta(k, \sigma_{b_i=1})$   
  ELSE  
     $\delta'(k, \sigma) = \delta(k, \sigma_{b_i=0})$

**Fig. 2.** Projection algorithm of Theorem 1

### 3.1 Image Computation for $x'_i = c$ Updates

For each category of updates we discuss both pre-condition and post-condition computations. For each case we simplify the formulas  $\phi_{\text{pre}(S,R)}$  and  $\phi_{\text{post}(S,R)}$  and discuss the computation of their automata representations.

#### Pre-condition Computation

$$\begin{aligned}
\phi_{\text{pre}(S,R)} &\Leftrightarrow \exists x'_1 \dots \exists x'_v. (\phi'_S \wedge \phi_R) \\
&\Leftrightarrow \exists x'_1 \dots \exists x'_v. (\phi'_S \wedge \text{guard}(R) \wedge \text{update}(R)) \\
&\Leftrightarrow (\exists x'_1 \dots \exists x'_v. (\phi'_S \wedge \text{update}(R))) \wedge \text{guard}(R) \\
&\Leftrightarrow (\exists x'_1 \dots \exists x'_v. (\phi'_S \wedge x'_i = c \wedge \bigwedge_{j \neq i} x'_j = x_j)) \wedge \text{guard}(R) \\
&\Leftrightarrow (\exists x'_i. (\phi_{S[x'_i \leftarrow x'_i]} \wedge x'_i = c)) \wedge \text{guard}(R) \\
&\Leftrightarrow (\exists x_i. (\phi_S \wedge x_i = c)) \wedge \text{guard}(R).
\end{aligned}$$

$\text{FA}(x_i = c)$  can be constructed in  $O(\log_2 c)$  time and has  $O(\log_2 c)$  states. Thus,  $\text{FA}(\phi_S \wedge x_i = c)$  has  $O(|\text{FA}(\phi_S)| \cdot \log_2 c)$  states. By Theorem 1, the size of  $\text{FA}(\exists x_i. (\phi_S \wedge x_i = c))$  and the time needed to compute it is  $O(|\text{FA}(\phi_S)| \cdot \log_2 c)$ .

#### Post-condition Computation

$$\begin{aligned}
\phi_{\text{post}(S,R)} &\Leftrightarrow (\exists x_1 \dots \exists x_v. (\phi_S \wedge \phi_R))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]} \\
&\Leftrightarrow (\exists x_1 \dots \exists x_v. (\phi_S \wedge \text{guard}(R) \wedge \text{update}(R)))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]} \\
&\Leftrightarrow (\exists x_1 \dots \exists x_v. (\phi_S \wedge \text{guard}(R) \wedge x'_i = c \wedge \bigwedge_{j \neq i} x'_j = x_j))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]} \\
&\Leftrightarrow (\exists x_i. (\phi_S \wedge \text{guard}(R))) \wedge x_i = c.
\end{aligned}$$

Unfortunately, we cannot use Theorem 1 in this case. We can compute  $\text{FA}(\exists x_i. (\phi_S \wedge \text{guard}(R)))$  from  $\text{FA}(\phi_S \wedge \text{guard}(R))$  by projection with a worst case exponential time and space complexity.

### 3.2 Image Computation for $x'_i = x_i + c$ Updates

Suppose  $\phi_S$  and  $guard(R)$  consist of atomic linear constraints of the form  $\phi_k : \sum_{i=1}^v a_{i,k} \cdot x_i \sim c_k, 1 \leq k \leq l$ , where  $\sim \in \{=, \neq, >, \geq, \leq, <\}$ , and Boolean connectives.

#### Pre-condition Computation

$$\begin{aligned} \phi_{pre(S,R)} &\Leftrightarrow (\exists x'_1 \dots \exists x'_v. (\phi'_S \wedge x'_i = x_i + c \wedge \bigwedge_{j \neq i} x'_j = x_j)) \wedge guard(R) \\ &\Leftrightarrow \phi_{S[x_i \leftarrow x_i + c]} \wedge guard(R) \Leftrightarrow \phi_{S[c_k \leftarrow c_k - a_{i,k} \cdot c]} \wedge guard(R). \end{aligned}$$

#### Post-condition Computation

$$\begin{aligned} \phi_{post(S,R)} &\Leftrightarrow (\exists x_1 \dots \exists x_v. (\phi_S \wedge guard(R) \wedge x'_i = x_i + c \wedge \\ &\quad \bigwedge_{j \neq i} x'_j = x_j))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]} \\ &\Leftrightarrow (\phi_S \wedge guard(R))_{[x_i \leftarrow x_i - c]} \Leftrightarrow (\phi_S \wedge guard(R))_{[c_k \leftarrow c_k + a_{i,k} \cdot c]}. \end{aligned}$$

It is clear that for both pre- and post-condition computation only the constant term changes in each atomic linear constraint. An algorithm that changes the constant term in an atomic equation is shown in Figure 3. The algorithm for inequations is similar. Note that the complexity of both algorithms is proportional to the number of new states introduced, which is possibly 0 or at most  $|c'|$ . These algorithms assume that atomic formulas are stored with the corresponding automata. In our implementation this is not the case, and we actually use the more general approach presented next.

### 3.3 Image Computation for $x'_i = \sum_{j=1}^v a_j \cdot x_j + c$ Updates

#### Pre-condition Computation

$$\begin{aligned} \phi_{pre(S,R)} &\Leftrightarrow (\exists x'_1 \dots \exists x'_v. (\phi'_S \wedge x'_i = \sum_{j=1}^v a_j \cdot x_j + c \wedge \bigwedge_{j \neq i} x'_j = x_j)) \wedge guard(R) \\ &\Leftrightarrow (\exists x'_i. (\phi_{S[x_i \leftarrow x'_i]} \wedge x'_i = \sum_{j=1}^v a_j \cdot x_j + c)) \wedge guard(R). \end{aligned}$$

Again we can use Theorem 1 to prove that existential variable elimination can be performed in linear time without increasing the automaton size. We use the algorithm in Figure 2 to compute  $FA(\exists x'_i. (\phi_{S[x_i \leftarrow x'_i]} \wedge x'_i = \sum_{j=1}^v a_j \cdot x_j + c))$ .

```

Input FA( $\sum_{i=1}^v a_i \cdot x_i = c$ ) = ( $K, \Sigma, \delta, e, F$ )
Output FA( $\sum_{i=1}^v a_i \cdot x_i = c'$ ) = ( $K', \Sigma', \delta', e', F'$ )

IF  $-c' \in K$  THEN
   $K' = K$   $\Sigma' = \Sigma$   $\delta' = \delta$   $e' = -c'$   $F' = F$ 
ELSE
   $K' = K \cup \{-c'\}$   $\Sigma' = \Sigma$   $\delta' = \delta$   $e' = -c'$   $F' = F$ 
  WHILE  $\exists k \in K', \sigma \in \Sigma'$  s.t.  $\delta'(k, \sigma) = \text{null}$  DO
    FOR ALL  $\sigma = (b_1, \dots, b_v) \in \Sigma'$  DO
      IF  $l := (\sum_{i=1}^v a_i \cdot b_i + k)/2 \in \mathbb{Z}$  THEN
         $K' := K' \cup \{l\}$ 
         $\delta'(k, \sigma) := l$ 
      ELSE
         $\delta'(k, \sigma) := \text{sink}$ 

```

**Fig. 3.** Algorithm for changing the constant term in equations

## Post-condition Computation

$$\begin{aligned}
\phi_{post(S,R)} &\Leftrightarrow (\exists x_1 \dots \exists x_v. (\phi_S \wedge guard(R) \wedge x'_i = \sum_{j=1}^v a_j \cdot x_j + c \wedge \\
&\quad \bigwedge_{j \neq i} x'_j = x_j))_{[x'_1 \leftarrow x_1, \dots, x'_v \leftarrow x_v]} \\
&\Leftrightarrow (\exists x_i. (\phi_S \wedge guard(R) \wedge x'_i = \sum_{j=1}^v a_j \cdot x_j + c))_{[x'_i \leftarrow x_i]}.
\end{aligned}$$

Note that in this case, Theorem 1 applies only when  $a_i$  is an odd integer and in that case we can use the algorithm in Figure 2.

## 4 Implementation

A problem with the FA representation for arithmetic constraints is the size of the transition function, since the number of transitions from each state is exponential on  $v$ , the number of integer variables. Hence, it is impractical to store the transition function as a table. Actual implementations use different solutions to this problem. We have used the approach used in MONA [14]. MONA is an automata package that uses BDDs [6] to store the transition function. In particular, for each FA state  $n$ , there is a BDD representing the function  $\delta(n, (b_1, \dots, b_v))$ . The terminal nodes are also FA states and internal nodes can be shared. To evaluate  $\delta(n, (b_1, \dots, b_v))$  one should start at the root of the BDD for state  $n$  and move from node to node depending on the values of  $b_1, \dots, b_v$  until a leaf node is reached. That leaf node corresponds to the state  $\delta(n, (b_1, \dots, b_v))$ . Since BDDs are

a canonical representation for Boolean functions, given a fixed variable ordering, the size of the transition relation can be kept minimal, e.g., variables with zero coefficients do not appear in the BDD representing the transition function. We can also prove that the size of the BDD is linear in the number of variables  $v$  and not exponential [2].

The algorithm of Theorem 1 is linear in the number of transitions. When the transition function is represented as a BDD we would like the algorithm to be linear in the number of BDD nodes in the automaton. For the case of pre-condition computation, this is feasible for a new category of updates:  $x'_i = \sum_{j=1}^i a_j \cdot x_j + c$ , i.e. updates where the new value of  $x_i$  depends on the old value of itself or variables with smaller indices. This new category includes the original categories 1 and 2. For a system with  $v$  variables, we fix the order of the variables in the BDDs to be  $x_1, x'_1, \dots, x_v, x'_v$ . In  $\text{FA}(\phi_{S[x_i \leftarrow x'_i]} \wedge x'_i = \sum_{j=1}^i a_j \cdot x_j + c)$ , according to Theorem 1, we know that at least one of the children of any node for variable  $x'_i$  points to the *sink* state. Consequently, we can compute  $\text{FA}(\exists x'_i. (\phi_{S[x_i \leftarrow x'_i]} \wedge x'_i = \sum_{j=1}^i a_j \cdot x_j + c))$  by visiting all nodes for variable  $x_i$  and delete one of the children that goes to the *sink* state. Every BDD node is visited once, thus the whole operation can be performed in time linear in the total number of BDD nodes in the automaton.

The BDD representation of the FA transition function also allows efficient handling of boolean formulas [2]. To accommodate boolean variables, we encode *false* with  $0(0 \cup 1)^*$  and *true* with  $1(0 \cup 1)^*$ . This way, in an automaton that represents a composite formula with both boolean and integer variables, only the BDD rooted at the initial state will contain nodes that depend on the boolean variables. All other BDDs will contain only nodes for the integer variables and thus their size is independent of the number of boolean variables. In other words, the BDD rooted at the initial state evaluates the boolean part of the formula and the rest of the automaton evaluates the integer part.

Given a formula  $\phi$  containing both boolean and integer variables, we define a boolean subformula of  $\phi$  to be either a boolean variable appearing in  $\phi$ , a negated boolean variable, a constant *true* or *false*, or two boolean subformulas connected by a logical connective ( $\wedge, \vee$ , etc) in  $\phi$ . A maximal boolean subformula of  $\phi$  is a boolean subformula of  $\phi$  that is not contained in any other boolean subformula of  $\phi$ .

Now suppose that  $\phi$  is a general formula containing distinct maximal boolean subformulas ( $B_1, \dots, B_n$ ) and distinct atomic linear integer arithmetic constraints ( $P_1, \dots, P_m$ ) combined with boolean connectives. We can prove that the total size of the BDD representing the transition function of  $\text{FA}(\phi)$  is  $O(\prod_{i=1}^n |\text{FA}(B_i)| + \prod_{i=1}^m (|\text{FA}(P_i)| + 1))$ , where  $|\text{FA}(B_i)|$  and  $|\text{FA}(P_i)|$  are the sizes (in BDD nodes) of the automata representing  $B_i$  and  $P_i$  respectively [2].

## 5 Experiments

We integrated the construction algorithms in [3,2] as well as the pre- and post-condition computation algorithms presented in this paper to an infinite state

CTL model checker called Action Language Verifier (ALV) [7] built on top of the Composite Symbolic Library [21]. The Composite Symbolic Library uses an object-oriented design to combine different symbolic representations [21]. In our experiments we compare the efficiency of our implementation with BRAIN [17] that uses Hilbert's basis as canonical representation for arithmetic constraints, and LASH [1] that uses the automata representation. To make the comparison with LASH fair, we integrated the automata construction and manipulation algorithms used in LASH to the Action Language Verifier.

We experimented with a large set of examples taken from 1) The Action Language Verifier distribution at: <http://www.cs.ucsb.edu/~bultan/composite/> and 2) The BRAIN distribution at: <http://www.cs.man.ac.uk/~voronkov/BRAIN/>. All the examples used in our experiments and the executables of the tools are available at: <http://www.cs.ucsb.edu/~bar/image>. We obtained the experimental results on a SUN ULTRA 10 work station with 768 Mbytes of memory, running SunOs 5.7. The results are presented in Table 1. Time measurements appear in seconds. Entries of  $\uparrow\uparrow$  mean that the computation was aborted because the memory limit was exceeded. Entries of  $\uparrow$  mean that the computation was aborted at 12000 seconds because no significant progress was made. For the automata representation used in ALV we also recorded the size (i.e., number of BDD nodes) of the largest automaton computed.

The experimental results show that the automata representation used in LASH is not efficient. There are two main reasons for this inefficiency. First, LASH stores the transition function of automata explicitly as opposed to the multi-terminal BDD representation used in MONA. Second, LASH implements the image computation using a standard automata projection algorithm which has a worst case exponential complexity, as opposed to the polynomial time image computations proposed here.

Problem instances can be categorized in three groups:

1. Pure integer problems (CSM, incdec, bigjava, consistencyprot and consprod)
2. Integer problems with invariants (those with the suffix inv)
3. Problems with both boolean and integer variables (bakery and barber)

Except for the consistency protocol problem instance, it is clear that ALV is faster than BRAIN for groups 2 and 3 and BRAIN is faster only for problems in group 1. The problems with invariants are obtained from the original problems by adding invariants. A typical invariant has the form  $x_1 + \dots + x_k < m$ , where  $m$  is a natural number. Such invariants essentially bound the variables  $x_1, \dots, x_k$  to a finite region. The presence of finite domain variables causes a problem for BRAIN, because the size of the Basis (the canonical representation used in BRAIN) can grow exponentially. On the other hand, systems with finite domain variables can be efficiently encoded by automata with transition functions stored as BDDs.

There are several advantages of the automata representation of arithmetic formulas over the Hilbert's basis representation used in BRAIN:

**Table 1.** Experimental results. Time measurements appear in seconds. Max size is the number of BDD nodes of the largest automaton computed for each problem instance

Problem Instance	BRAIN	ALV		LASH
	time	time	max size	time
CSM4	3.76	99.35	25910	↑
CSM6	25.01	540.88	110796	↑
CSM8	128.54	1772.85	238739	↑
CSM10	494.03	4809.13	484249	↑
CSM12	1644.33	9676.81	839870	↑
CSMinv10	0.93	0.58	485	217.76
CSMinv20	3.57	0.90	606	282.49
CSMinv30	9.59	1.09	727	416.46
CSMinv40	20.71	1.20	727	458.48
CSMinv50	38.58	1.45	910	601.21
incdec	195.48	2792.54	258945	↑
incdecinv	24.79	4.67	1194	↑
bakery3	0.35	0.38	509	10.65
bakery4	14.82	9.83	8762	244.67
bakery5	1107.75	577.45	230906	↑
barber4	0.74	0.25	76	11.79
barber8	52.05	0.78	136	27.01
barber12	14669.80	1.81	212	53.41
bigjava	11244.60	↑↑	↑↑	↑
bigjavainv	2641.05	82.33	6157	↑
bigjavainv1	30615.20	1160.09	45114	↑
consistencyprot	1.09	24.28	15049	↑
consistencyprotinv	7.75	59.38	31453	↑
consistencyprotinv1	0.05	0.16	212	182.84
consprod	11346.40	↑↑	↑↑	↑
consprodinv	1.27	0.66	253	↑

1. Automata can handle a larger class of systems, namely all transition systems representable by Presburger arithmetic formulas. BRAIN cannot handle quantified formulas or divisibility constraints.
2. For the class of systems for which BRAIN provides polynomial time image computation we prove polynomial bounds for the automata representation and give the algorithms. Even for problems in group 1 for which BRAIN is faster than ALV, the speedup achieved by BRAIN seems to be constant, which is what we would expect given that both techniques have equally efficient image computations. In particular, for problem CSM, ALV scales better even though BRAIN is faster.
3. The automata representation can handle forward image computation and solve problems for which the backward fixpoint computation does not converge, but the forward computation does. Such problems are not solvable using BRAIN. For example, we can verify mutual exclusion and starvation

freedom properties for the ticket mutual exclusion protocol [8] using forward fixpoint computations, whereas this is not possible for BRAIN.

4. The automata representation can be combined with an efficient encoding of boolean and enumerated variables, however it is not clear if this could be done efficiently with the Hilbert's basis technique. In BRAIN specification of the problems in group 3, boolean and enumerated variables have been mapped to integers. The experimental results indicate the inefficiency of this mapping, which becomes more apparent when the problem size increases.
5. Using the automata representation we can perform full CTL verification, whereas BRAIN can only verify invariants. For example, for the bakery protocol we can verify liveness properties of the form:  $AG(pc = try \Rightarrow AF(pc = cs))$ , while BRAIN cannot.

However, on pure integer problems with large number of variables, BRAIN outperforms ALV. We plan to investigate if this is due to the efficiency of the Hilbert's Basis representation or due to the fixpoint computation algorithm used in BRAIN.

## 6 Conclusion

In this paper we show that for a common class of infinite state systems represented by linear arithmetic constraints, image computations can be done efficiently without an exponential blow up. We give algorithms for efficient image computations for updates that are expressed as linear equalities based on an automata encoding of the states of the system. We implemented these algorithms and experiments show that they improve the efficiency of automata based representations significantly. Experiments also indicate that, in a lot of cases, automata encoding with the proposed image computations is as efficient as other more restrictive canonical representations.

The results in this paper can also be used to show that image computation on bounded arithmetic constraints represented by BDDs can be done in polynomial time for a class of arithmetic constraints. We plan to develop algorithms for the bounded case based on the BDD encodings of arithmetic constraints presented in [4].

## References

1. The Liège Automata-based Symbolic Handler (LASH). Available at <http://www.montefiore.ulg.ac.be/~boigelot/research/lash/>.
2. Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science*. To appear.
3. Constantinos Bartzis and Tevfik Bultan. Automata-based representations for arithmetic constraints in automated verification. In *Proceedings of the Seventh International Conference on Implementation and Application of Automata*, 2002.



4. Constantinos Bartzis and Tevfik Bultan. Construction of efficient BDDs for bounded arithmetic constraints. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.
5. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming - CAAP'96*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, April 1996.
6. R. Bryant. Graph-based algorithms for boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.
7. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.
8. Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.
9. G. Delzanno and T. Bultan. Constraint-based verification of client-server protocols. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, 2001.
10. Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.
11. N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Proceedings of the 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212. Springer-Verlag, 2001.
12. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.
13. T. A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
14. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, 2001.
15. K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.
16. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis, 1992.
17. Tatiana Rybina and Andrei Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 400–411, 2002.
18. P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of the Static Analysis Symposium*, 1995.
19. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 1–19. Springer, April 2000.
20. T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 169–179, July 2002.
21. T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.

# Thread-Modular Abstraction Refinement<sup>\*</sup>

Thomas A. Henzinger<sup>1</sup>, Ranjit Jhala<sup>1</sup>, Rupak Majumdar<sup>1</sup>, and Shaz Qadeer<sup>2</sup>

<sup>1</sup> University of California, Berkeley

<sup>2</sup> Microsoft Research, Redmond

**Abstract.** We present an algorithm called TAR (“Thread-modular Abstraction Refinement”) for model checking safety properties of concurrent software. The TAR algorithm uses thread-modular assume-guarantee reasoning to overcome the exponential complexity in the control state of multithreaded programs. Thread modularity means that TAR explores the state space of one thread at a time, making assumptions about how the environment can interfere. The TAR algorithm uses counterexample-guided predicate-abstraction refinement to overcome the usually infinite complexity in the data state of C programs. A successive approximation scheme automatically infers the necessary precision on data variables as well as suitable environment assumptions. The scheme is novel in that transition relations are approximated from above, while at the same time environment assumptions are approximated from below. In our software verification tool BLAST we have implemented a fully automatic race checker for multithreaded C programs which is based on the TAR algorithm. This tool has verified a wide variety of commonly used locking idioms, including locking schemes that are not amenable to existing dynamic and static race checkers such as ERASER or WARLOCK.

## 1 Introduction

Many model-checking tools for software analysis fall into two categories. The first category of tools, pioneered by SPIN [15], relies on the user-guided representation of a program as an abstract model, which is then checked for the desired properties. Several tools in this category support model extraction from code [7,13,16,21], but they still rely on the user to define the granularity of the abstraction. The second category of tools, pioneered by SLAM [4], automatically refines an extracted model to the necessary precision. While highly successful, both approaches have their limitations. The SLAM approach is fully automatic but, so far, has been limited to sequential programs. This is unfortunate, because the traditional strength of model checking lies in the analysis of concurrent systems, such as multithreaded programs, in which errors are notoriously difficult to reproduce. The SPIN approach is well-suited for concurrency but relies on the insights of the user to find appropriate abstractions. This is also unfortunate, because the other traditional strength of model checking is its “push-button”

---

<sup>\*</sup> This work was supported in part by the NSF grants CCR-0085949 and CCR-0234690, the DARPA grant F33615-00-C-1693, and the MARCO grant 98-DT-660.

characteristics, which in the realm of hardware verification has enabled the routine application of model-checking tools. We present an algorithm and tool that for the first time uses fully automatic abstraction refinement on concurrent, multithreaded programs.

The first ingredient of our algorithm is *thread-modular assume-guarantee reasoning*. The main source of complexity in multithreaded programs is the interaction between threads that operate on shared data. A model checker must analyze all possible interleavings of the actions of the various threads, resulting in the dreaded state-explosion problem. One method for controlling state explosion is thread-modular reasoning, first proposed by Jones [17] and first implemented in the CALVIN checker [9,11] for multithreaded Java programs. To check a 2-threaded program  $T_1||T_2$ , CALVIN allows the programmer to specify suitable abstractions  $G_1$  and  $G_2$  for the transition relations  $T_1$  of thread 1 and  $T_2$  of thread 2, and then separately analyzes  $T_1||G_2$  and  $G_1||T_2$ . This reasoning is “thread-modular” if the abstractions  $G_1$  and  $G_2$  constrain only the shared variables, and consequently the analyzed systems  $T_1||G_2$  and  $G_1||T_2$  each depend on the private variables of at most one thread. The abstraction  $G_2$  is an environment assumption for thread 1, and  $G_1$  constrains the environment of thread 2. Thread-modular reasoning, therefore, is a form of assume-guarantee reasoning [1,2], which is sound for safety properties:

If no error states are reachable in  $T_1||G_2$  nor in  $G_1||T_2$ , and  $T_1||G_2 \subseteq G_1$  and  $G_1||T_2 \subseteq G_2$ , then no error states are reachable in  $T_1||T_2$ . (AG)

While thread-modular reasoning is not complete for safety properties [18], it suffices for establishing the safety of many concurrency-control mechanisms commonly used in multithreaded programming [9]. The main obstacle to thread-modular reasoning is the annotation burden involved in specifying the environment assumptions  $G_1$  and  $G_2$ . Although these assumptions can be inferred automatically for finite-state systems [10], in the presence of unbounded data they have to be supplied manually for tools such as CALVIN.

This limitation motivates the second ingredient of our algorithm, *counterexample-guided predicate-abstraction refinement*. Many relations between data values are irrelevant for the task of proving a desired property, but in general it is difficult to divine the relevant relations (“predicates” [12]). Indeed, the process of manually finding a suitable abstract model, which is neither too detailed to choke the model checker nor too coarse to invalidate the specification, often dominates the verification effort. This process has been automated by the abstract-check-refine paradigm [3,6]: one starts with a very coarse abstraction, and if the model checker finds an abstract error trace that has no concrete counterpart, then one uses that trace to automatically refine the abstraction, and repeats the process. In particular, a theorem prover can be used to automatically discover predicates that rule out spurious counterexamples and thus are good candidates for abstraction refinement [14]. This approach to automatic abstraction refinement has been very successful for sequential programs [4], but has its problems when dealing with concurrency. Suppose that we try to find appropriate abstractions of the two thread transition relations  $T_1$  and  $T_2$ , as well

as abstractions of the two environment assumptions  $G_1$  and  $G_2$ , by iteratively approximating all four relations from above until condition (AG) holds. We will see in Section 2 that this approximation scheme fails in many cases of practical interest: even if suitable thread-modular assumptions  $G_1$  and  $G_2$  exist, one may end up with approximations  $t_1 \supseteq T_1$ ,  $t_2 \supseteq T_2$ ,  $g_1 \supseteq t_1 || g_2$ , and  $g_2 \supseteq g_1 || t_2$  such that error states are reachable in  $t_1 || g_2$  or  $g_1 || t_2$ , but no new predicates on the shared variables can be discovered to remove these error traces. We overcome this hurdle by approximating environment assumptions from *below*, rather than above. The algorithm TAR (“Thread-modular Abstraction Refinement”) operates with approximations of four relations:  $t_1 \supseteq T_1$  and  $t_2 \supseteq T_2$  are traditional overapproximations of the two thread transition relations, and are successively refined by the addition of new predicates. The approximations  $g_1$  and  $g_2$  of the environment assumptions, however, start out empty (which corresponds to there being no environment) and are successively weakened until  $g_1 \supseteq t_1 || g_2$  and  $g_2 \supseteq g_1 || t_2$ . By interleaving the strengthening of the  $t_i$  relations and the weakening of the  $g_i$  relations in an appropriate way, we can prove that if environment assumptions  $G_1$  and  $G_2$  that satisfy condition (AG) exist, then they will be found in the limit, that is, on finite state spaces they are guaranteed to be found in a finite number of iterations. In other words, the TAR algorithm provides a sound combination of thread-modular assume-guarantee reasoning and counterexample-guided abstraction refinement which, in the case of finite-state systems, is also complete in the sense that thread-modular environment assumptions are found if they exist.

We have implemented the TAR algorithm in the model checker BLAST [14], which was originally designed for the verification of sequential C programs. BLAST deals with all aspects of C, such as function calls, structures, and pointer aliasing, and uses an on-the-fly algorithm for integrated reachability analysis and predicate discovery (“lazy abstraction”). The TAR extension of BLAST is targeted to find *data races* in multithreaded C programs. Unlike much of the exposition in this paper, which considers for simplicity only two threads, the tool deals with any number of concurrent threads. Data races are states in which either two different threads update a shared data variable (as opposed to a lock variable), or one thread updates and another thread reads the variable. Race detection can therefore be formulated as a safety verification problem. We have focused on race detection for two reasons. First, race checking requires no code annotations or specifications from the user, and is therefore particularly appealing to practitioners. Second, the absence of race conditions is a prerequisite for establishing a variety of more complicated correctness requirements.

Section 2 presents the TAR algorithm in a generic setting, together with soundness and termination (completeness) arguments. Section 3 develops a version of the algorithm in the specific setting of race detection for multithreaded C programs. It gives a producer-consumer example for which the TAR algorithm proves the absence of race conditions, whereas all existing race checkers available to us, both dynamic (“lockset”-based) [19] and static (“type”-based) [5,8,20], report false positives. Section 4 briefly discusses the implementation of the TAR

algorithm in BLAST. Section 5 summarizes our experience with the tool. We have applied the race checker with success to a number of examples that capture a variety of synchronization idioms commonly used in operating systems and databases, including schemes where the same variable is protected at different times by different locks, which again lie outside the scope of applicability of existing race checkers. We have used BLAST to check several thousand lines of multithreaded C code, such as Linux and Windows device drivers, and have verified the absence of race conditions. In the case of one Windows driver, a race was found, and although benign, its inspection led to the discovery of a real concurrency error in the code.

## 2 A Thread-Modular Abstraction Refinement Algorithm

**Thread-modular reasoning.** A 2-threaded program  $\Pi = (Q, Q_0, T_1, T_2, E)$  consists of the following components. The state space has the form  $Q = S \times P_1 \times P_2$ , where  $S$  is the shared state space, and  $P_i$  is the private state space of thread  $i$ , for  $i = 1, 2$ . We write  $Q_i = S \times P_i$  for the state space of thread  $i$ . There is a set  $Q_0 \subseteq Q$  of initial states, a transition relation  $T_i \subseteq Q_i^2$  for each thread  $i$ , and a set  $E \subseteq Q$  of error states. Given a relation  $t \subseteq Q^2$ , a  $t$ -trace  $q_0, q_1, \dots, q_n$  is a finite sequence of states  $q_j \in Q$  such that (1)  $q_0 \in Q_0$  and (2)  $(q_j, q_{j+1}) \in t$  for all  $0 \leq j < n$ . A state  $q \in Q$  is  $t$ -reachable iff there is a  $t$ -trace whose last state is  $q$ . We write  $R(t)$  for the set of  $t$ -reachable states. Given two relations  $t_1 \subseteq Q_1^2$  and  $t_2 \subseteq Q_2^2$ , the interleaving  $(t_1 || t_2) \subseteq Q^2$  is the relation defined by  $((s, p_1, p_2), (s', p'_1, p'_2)) \in (t_1 || t_2)$  iff either  $((s, p_1), (s', p'_1)) \in t_1$  and  $p_2 = p'_2$ , or  $((s, p_2), (s', p'_2)) \in t_2$  and  $p_1 = p'_1$ . We write  $\sigma[t_i] \subseteq Q_i^2$  for the relation defined by  $((s, p_i), (s', p'_i)) \in \sigma[t_i]$  iff  $((s, \hat{p}_i), (s', \hat{p}'_i)) \in t_i$  for some private states  $\hat{p}_i, \hat{p}'_i \in P_i$ . The relation  $\sigma[t_i] \supseteq t_i$  on the state space of thread  $i$  conforms with  $t_i$  on the shared state, but updates the private state in arbitrary ways. We write  $\rho_i[t_1, t_2] \subseteq Q_i^2$  for the relation defined by  $((s, p), (s', p')) \in \rho_i[t_1, t_2]$  iff there is some state  $(s, p_1, p_2)$  that is  $(t_1 || t_2)$ -reachable and  $((s, p_1), (s', p'_1)) \in t_i$  for some  $p'_1$ . The relation  $\rho_i[t_1, t_2]$  is the restriction of  $\sigma[t_i]$  to the  $(t_1 || t_2)$ -reachable states.

The program  $\Pi$  is *safe* iff  $R(T_1 || T_2) \cap E = \emptyset$ . Safety means that there is no  $(T_1 || T_2)$ -trace that ends in an error state. The program  $\Pi$  is *safe in a strongly thread-modular way* iff  $R(T_1 || \sigma[T_2]) \cap R(\sigma[T_1] || T_2) \cap E = \emptyset$ . Strongly thread-modular safety means that there are no  $(T_1 || \sigma[T_2])$ - and  $(\sigma[T_1] || T_2)$ -traces that end in the same error state. In the strongly thread-modular case, suitable thread-modular environment assumptions  $\sigma[T_i]$  can be obtained noninductively, by existential quantification of the private variables of thread  $i$ . The general thread-modular case permits stronger environment assumptions, which may be obtained inductively by taking into account reachability information. Formally, the program  $\Pi$  is *safe in a thread-modular way* [9] iff there are environment assumptions  $G_1$  and  $G_2$  such that  $G_1 \supseteq \rho_1[T_1, G_2]$  and  $G_2 \supseteq \rho_2[G_1, T_2]$  and  $R(T_1 || G_2) \cap R(G_1 || T_2) \cap E = \emptyset$ . If a program is safe in a thread-modular way, then it can be proved without considering the product transition relation  $T_1 || T_2$ ;

it suffices to reason about thread 1 together with an environment assumption about thread 2 that concerns only the shared state, and vice versa.

**Proposition 1a.** Every program that is safe in a strongly thread-modular way is safe in a thread-modular way. Every program that is safe in a thread-modular way is safe.  $\square$

**Proposition 1b.** There is a program that is safe but not in a thread-modular way. There is a program that is safe in a thread-modular way but not in a strongly thread-modular way.  $\square$

The second part of Proposition 1a follows by circular assume-guarantee reasoning [2]. For Proposition 1b, the existence of a program that is safe but not in a thread-modular way follows from the incompleteness of thread-modular reasoning [10]. The following example shows a program that is safe in a thread-modular way but not in a strongly thread-modular way.

**Example 1.** Consider a 2-threaded program with shared variables  $m$  and  $x$ , both initially 0. The variable  $m$  represents a lock; its value is 0 if no thread holds the lock, and  $i$  if thread  $i$  holds it. Thread  $i$  executes the following code:

`while (1) do { acquire(m); x=i; release(m); }`

We model the program counter of thread  $i$  by a private variable  $pc_i$  initialized to 0. The transition relation  $T_i$  of thread  $i$  is

$$(pc_i = 0 \wedge pc'_i = 1 \wedge m' = m \wedge x' = x) \vee (pc_i = 1 \wedge m = 0 \wedge pc'_i = 2 \wedge m' = 1 \wedge x' = x) \\ \vee (pc_i = 2 \wedge pc'_i = 3 \wedge m' = m \wedge x' = i) \vee (pc_i = 3 \wedge pc'_i = 0 \wedge m' = 0 \wedge x' = x).$$

The error states  $E$  are those in which a data race occurs:  $pc_1 = 2 \wedge pc_2 = 2$ . For  $i = 1, 2$ :

$$\sigma[T_i] : (m' = m \wedge x' = x) \vee (m = 0 \wedge m' = i \wedge x' = x) \vee \\ (m' = m \wedge x' = i) \vee (m' = 0 \wedge x' = x) \\ \rho_i[T_1, T_2] : (m \neq i \wedge m' = m \wedge x' = x) \vee (m = 0 \wedge m' = i \wedge x' = x) \vee \\ (m = i \wedge m' = m \wedge x' = i) \vee (m = i \wedge m' = 0 \wedge x = i \wedge x' = x)$$

The error state  $pc_1 = 2 \wedge x = 0 \wedge m = 0 \wedge pc_2 = 2$  is reachable by the transition sequences  $T_1, T_1, \sigma[T_2]$  and  $T_2, T_2, \sigma[T_1]$ ; hence the program is not safe in a strongly thread-modular way. However, the program is safe in a thread-modular way, because the extra conjuncts in the last clauses of  $\rho_1[T_1, T_2]$  and  $\rho_2[T_1, T_2]$  ensure that all states in  $R(T_1 || \rho_2[T_1, T_2])$  with  $pc_1 = 2$  have  $m = 1$ , and all states in  $R(\rho_1[T_1, T_2] || T_2)$  with  $pc_2 = 2$  have  $m = 2$ , thus making the intersection empty.  $\square$

**Abstraction refinement.** To establish the correctness and termination of our algorithm, we treat abstraction refinement as a black box (oracle). A *refinement function*  $\Theta$  is given two relations  $(t_1, t_2) \subseteq Q_1^2 \times Q_2^2$  such that (1) either  $T_1 \subseteq t_1$  and  $T_2 \subset t_2$ , or  $T_1 \subset t_1$  and  $T_2 \subseteq t_2$ , and (2) there are  $(t_1 || \sigma[t_2])$ - and  $(\sigma[t_1] || t_2)$ -traces that end in error states, but none of them is a  $(T_1 || T_2)$ -trace. The function  $\Theta$  returns a pair  $(r_1, r_2) \subseteq Q_1^2 \times Q_2^2$  of relations such that (1)  $T_i \subseteq r_i \subseteq t_i$  for  $i = 1, 2$ , and (2) either  $r_1 \subset t_1$  or  $r_2 \subset t_2$ . Our implementation of  $\Theta$  will look

at the reasons why the abstract counterexamples (i.e., traces to error states) cannot be concretized, and will add new predicates that rule out some abstract counterexamples by removing at least one transition from  $t_1$  or  $t_2$ , thus obtaining the new, refined abstract transition relations  $r_1$  and  $r_2$ .

*Naive abstraction refinement.* Abstraction refinement usually proceeds from a very coarse overapproximation of the transition relations, which are successively refined until either all abstract counterexamples are ruled out, or a concrete counterexample is found. In our setting, this can be accomplished by the following algorithm.

**Algorithm A.** Initially  $t_i := Q_i^2$  for  $i = 1, 2$ . Loop:

If  $R(t_1||t_2) \cap E = \emptyset$ , then return “safe.” If some  $(t_1||t_2)$ -trace that ends in an error state is a  $(T_1||T_2)$ -trace, then return “unsafe.” Let  $(t_1, t_2) := \Theta(t_1, t_2)$ .

Algorithm A computes with relations on the product space  $S \times P_1 \times P_2$ . A thread-modular algorithm computes only with relations on  $S \times P_1$  and  $S \times P_2$ . Algorithm B is a thread-modular version of A. It uses overapproximations  $t_i$  of  $T_i$ , and also thread-modular environment assumptions  $g_i \subseteq Q_i^2$ , for  $i = 1, 2$ .

**Algorithm B.** Initially  $t_i := Q_i^2$  and  $g_i := Q_i^2$  for  $i = 1, 2$ . Loop:

If  $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$ , then return “safe.” If some  $(t_1||g_2)$ - or  $(g_1||t_2)$ -trace that ends in an error state is also a  $(T_1||T_2)$ -trace, then return “unsafe.” If  $(t_1, t_2) = (T_1, T_2)$ , then return “don’t know.” Let  $(t_1, t_2) := \Theta(t_1, t_2)$  and let  $(g_1, g_2) := (\sigma[t_1], \sigma[t_2])$ .

**Proposition 2a.** If Algorithm B returns “safe,” then the program is safe in a strongly thread-modular way. If Algorithm B returns “unsafe,” then the program is not safe.  $\square$

**Proposition 2b.** If the state space  $Q$  is finite, then Algorithm B terminates. If additionally, the program is safe in a strongly thread-modular way, then Algorithm B terminates returning “safe.”  $\square$

In other words, Algorithm B succeeds exactly for the (finite-state) programs that are safe in a strongly thread-modular way; if the state space is finite, but the program is not safe in a strongly thread-modular way, then the algorithm always terminates with either “unsafe” or “don’t know.” Unfortunately, as Example 1 showed, standard locking schemes are not strongly thread-modular. In particular, Algorithm B terminates with “don’t know” on Example 1. To find the environment assumptions for programs that are thread-modular but not in a strong way, such as Example 1, we need to use a different approach.

*Thread-modular abstraction refinement.* The correctness and termination of Algorithms A and B were straightforward to prove, because both transition relations  $(t_1$  and  $t_2)$  and both environment assumptions  $(g_1$  and  $g_2)$  were approximated from above. The following algorithm approximates the transition

**Table 1.** Running Algorithm TAR on Example 1.**Iteration 1**

$t_1 : (pc_1 = 0 \wedge pc'_1 = 1) \vee (pc_1 = 1 \wedge pc'_1 = 2) \vee (pc_1 = 2 \wedge pc'_1 = 3) \vee (pc_1 = 3 \wedge pc'_1 = 0)$   
 $t_2 : (pc_2 = 0 \wedge pc'_2 = 1) \vee (pc_2 = 1 \wedge pc'_2 = 2) \vee (pc_2 = 2 \wedge pc'_2 = 3) \vee (pc_2 = 3 \wedge pc'_2 = 0)$   
 $g_1 : \text{false}, g_2 : \text{false}$

**Iteration 2**

$t_1 : (pc_1 = 0 \wedge pc'_1 = 1 \wedge m' = m) \vee (pc_1 = 1 \wedge pc'_1 = 2 \wedge m = 0 \wedge m' = 1) \vee$   
 $(pc_1 = 2 \wedge pc'_1 = 3 \wedge m' = m) \vee (pc_1 = 3 \wedge pc'_1 = 0 \wedge m' = 0)$   
 $t_2 : (pc_2 = 0 \wedge pc'_2 = 1 \wedge m' = m) \vee (pc_2 = 1 \wedge pc'_2 = 2 \wedge m = 0 \wedge m' = 2) \vee$   
 $(pc_2 = 2 \wedge pc'_2 = 3 \wedge m' = m) \vee (pc_2 = 3 \wedge pc'_2 = 0 \wedge m' = 0)$   
 $g_1 : \text{false}, g_2 : \text{false}$

**Iteration 3**

$t_1, t_2 : \text{same as in iteration 2}$   
 $g_1 : (m' = m) \vee (m = 0 \wedge m' = 1) \vee (m = 1 \wedge m' = 0)$   
 $g_2 : (m' = m) \vee (m = 0 \wedge m' = 2) \vee (m = 2 \wedge m' = 0)$

relations from above, but approximates the environment assumptions from below, and stops with success only once the successively coarsened environment assumptions move above the successively refined transition relations.

**Algorithm TAR.** Initially  $t_i := Q_i^2$  and  $g_i := \emptyset$  for  $i = 1, 2$ . Loop:

If  $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$  and  $\rho_1[t_1, g_2] \subseteq g_1$  and  $\rho_2[g_1, t_2] \subseteq g_2$ , then return “safe.” If some  $(t_1||g_2)$ - or  $(g_1||t_2)$ -trace that ends in an error state is also a  $(T_1||T_2)$ -trace, then return “unsafe.” If  $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$ , then let  $(g_1, g_2) := (\rho_1[t_1, g_2], \rho_2[g_1, t_2])$ ; otherwise, if  $(t_1, t_2) = (T_1, T_2)$ , then return “don’t know,” else let  $(t_1, t_2) := \Theta(t_1, t_2)$  and  $(g_1, g_2) := (\emptyset, \emptyset)$ .

**Theorem 3a.** If Algorithm TAR returns “safe,” then the program is safe in a thread-modular way. If Algorithm TAR returns “unsafe,” then the program is not safe.  $\square$

This is proved by circular assume-guarantee reasoning, because the environment assumption of each thread must be discharged against the abstract (overapproximate) transition relation of the other thread.

**Theorem 3b.** If the state space  $Q$  is finite, then Algorithm TAR terminates. If additionally, the program is safe in a thread-modular way, then Algorithm TAR terminates returning “safe.”  $\square$

In other words, Algorithm TAR succeeds exactly for the (finite-state) programs that are safe in a thread-modular way. It automatically finds suitable thread-modular environment assumptions, provided they exist. If the state space is finite but the program is not safe in a thread-modular way, then the algorithm always terminates with either “unsafe” or “don’t know.”

**Algorithm TAR on Example 1.** The initial abstraction contains the program counter of each thread (predicates of the form  $pc_i = 1$ ,  $pc_i = 2$ , etc.). The resulting existential overapproximations of  $T_1$  and  $T_2$  are  $t_1$  and  $t_2$ . In



this abstraction, the error state  $pc_1 = 2 \wedge x = 0 \wedge m = 0 \wedge pc_2 = 2$  is reachable. The counterexample analysis reveals that we should track the predicates  $m = 0$ ,  $m = 1$ , and  $m = 2$  (note that  $m' = m$  is shorthand for  $(m = 0 \Rightarrow m' = 0) \wedge (m = 1 \Rightarrow m' = 1) \wedge (m = 2 \Rightarrow m' = 2)$ ). At the end of iteration 2, there is no counterexample, because  $R(t_1||g_2) \cap R(g_1||t_2) \cap E = \emptyset$ . However, the test  $\rho_1[t_1, g_2] \subseteq g_1$  fails, because  $g_1$  is *false*, and  $\rho_1[t_1, g_2]$  is  $(m' = m) \vee (m = 0 \wedge m' = 1) \vee (m = 1 \wedge m' = 0)$ . We update  $g_1$  to  $\rho_1[t_1, g_2]$  and  $g_2$  to  $\rho_2[g_1, t_2]$  for iteration 3. In iteration 3, again there is no error, as  $R(t_i||g_{3-i})$  implies  $pc_i = 2 \Rightarrow m = i$ , for  $i = 1, 2$ . Moreover  $\rho_1[t_1, g_2] = g_1$  and  $\rho_2[g_1, t_2] = g_2$ . Thus the  $g_i$ 's are suitable environment assumptions, and the program is safe (in a thread-modular way).  $\square$

### 3 Race Detection

**Data races.** We now define the race-detection problem on a 2-threaded program  $\Pi = (Q, Q_0, T_1, T_2, E)$ . For the remainder of the paper,  $i$  ranges over  $\{1, 2\}$ . The program has a set *Shared* of variables visible to both threads, and sets *Private<sub>i</sub>* of variables visible only to thread  $i$ . Hence the shared (resp. private) state space  $S$  (resp.  $P_i$ ) is the set of all valuations to the variables in *Shared* (resp. *Private<sub>i</sub>*). The program counter of thread  $i$  is included in *Private<sub>i</sub>*. For each variable  $x \in \text{Shared}$ , let  $\text{Write}_i(x) \subseteq Q_i$  (resp.  $\text{Read}_i(x) \subseteq Q_i$ ) denote the set of states from which thread  $i$  writes (resp. reads)  $x$ . These states can be computed by a simple syntactic analysis of the program. For instance, in the program of Example 1,  $\text{Write}_i(x)$  is defined by the predicate  $pc_i = 2$ , and  $\text{Read}_i(x)$  is *false*. The *race-detection problem* for a variable  $x \in \text{Shared}$  asks if a state in  $E_x = \cup_{i \in \{1, 2\}} ((\text{Write}_i(x) \cup \text{Read}_i(x)) \cap \text{Write}_{3-i}(x))$  is  $(T_1||T_2)$ -reachable. Intuitively, there is a data race on the variable  $x$  if the program can reach a state in which both threads have enabled actions that access (read or write)  $x$ , and at least one of these accesses is a write. For the program of Example 1, the set  $E_x$  is  $pc_1 = 2 \wedge pc_2 = 2$ .

**Havoc abstractions.** We have implemented a variant of Algorithm TAR to check for data races: we further approximate each environment assumption  $g_i$  by predicates that use the domain (first coordinate) of the relation  $g_i$ , but ignore the effect of a transition (second coordinate). For each thread  $i$ , the algorithm maintains a *havoc* map  $h_i$  on *Shared*, such that  $h_i(x) \subseteq Q_i$  for every variable  $x \in \text{Shared}$ . Every havoc assumption  $h_i(x)$  gives an approximation of the set of states from which thread  $i$  can write to  $x$ . The environment assumption  $g_i$  induced by the havoc map  $h_i$  may change  $x$  to any arbitrary value (“havoc  $x$ ”) in a shared state  $s$  if there is some private state  $p_i$  such that  $(s, p_i) \in h_i(x)$ ; otherwise  $g_i$  leaves  $x$  unchanged. The initial havoc map assigns to every shared variable the empty set (represented by the predicate *false*). Hence, each thread initially assumes that the other threads do not modify any shared variable. Each iteration of Algorithm TAR updates the havoc map as follows. Whenever the overapproximations  $t_1$  and  $t_2$  of the thread transition relations are refined

<pre> thread Producer 1': while (1) { 2':   while (flag != 0) {}; 3':   data = get_new_data(); 4':   flag = 1;     } </pre>	<pre> thread Consumer 1:  while (1) { 2:    while (flag != 1) {}; 3:    copy = data; 4:    flag = 0;     } </pre>
---	---

**Fig. 1.** A producer-consumer system.

(by  $\Theta$ ), then all havoc assumptions are reinitialized to *false*. Otherwise,  $h_1(x)$  (resp.  $h_2(x)$ ) is updated to  $R(t_1||g_2) \cap Write_1(x)$  (resp.  $R(g_1||t_2) \cap Write_2(x)$ ).

**Example 1 with havoc.** In Example 1,  $Write_i(m)$  is  $(pc_i = 1 \wedge m = 0) \vee pc_i = 3$ . After iteration 2, the havoc assumption  $h_1(m)$  is  $(pc_1 = 1 \wedge m = 0) \vee (pc_1 = 3 \wedge m = 1)$ , meaning that when analyzing thread 2, the environment may change the value of  $m$  arbitrarily in states where  $m = 0 \vee m = 1$ .  $\square$

**Algorithm TAR with havoc.** The input is a multithreaded program, a set  $X \subseteq Shared$  of variables on which we check for data races, and a set of predicates that include control information about each thread.

**Initialization** (“Seed assumption”) Set the havoc assumption for each thread and each shared variable to *false*, i.e., the environment does nothing. The initial transition relation for each thread is the (existential) predicate abstraction of the thread’s transition relation w.r.t. the given predicates.

**Step 1** (“Reachability analysis”) Compute an abstraction  $R_i$  of the reachable states of each thread  $i$ , based on the current havoc assumptions about the behavior of the other threads, and the present set of predicates.

**Step 2** (“Counterexample analysis”) Check if the reach sets  $R_i$  computed in step 1 contain states with data races, i.e., check if  $R_1 \cap R_2 \cap E_x$  is nonempty for some variable  $x \in X$ . If the intersection is empty for all  $x \in X$ , then go to step 3. Otherwise, check if the abstract counterexample corresponds to a concrete program trace. If it does, then report the bug (“unsafe”); otherwise (a) discover new predicates that rule out the spurious counterexample and add them to the set of predicates, thus refining the transition relation of each thread, and (b) reset the havoc assumptions for each thread to *false*, and then go to step 1.

**Step 3** (“Discharge assumptions”) Check if the havoc assumptions are sound, i.e., for each thread  $i$  and variable  $x \in Shared$  check if  $h_i(x) \subseteq R_i \cap Write_i(x)$ . If so, report “safe”; otherwise update the havoc assumptions w.r.t. the present reach sets as discussed above, and then go to step 1.

**Example 2.** The 2-threaded program of Figure 1 has a **Producer** thread that produces new data items by writing to the variable **data**, and a **Consumer** thread that consumes the data items by reading from **data**. We illustrate how our algorithm verifies the absence of races on the shared variable **data**. The example is correct, because **Producer** writes to **data** only when it knows that **flag** is 0,

which happens after **Consumer** has removed the item that was put there last; and **Consumer** reads only when **Producer** is done with putting new data in, which it knows has happened when **flag** is set to 1. As the race-freedom depends on the value of the variable **flag**, and not on explicitly declared locks (as in Example 1, where  $m$  was a lock), existing static and dynamic race checkers report false positives for this program.

**Initialization** The havoc assumptions for both threads are set to *false*. In other words, when analyzing **Consumer** we assume that **Producer** does not affect the shared state in any way, and vice versa. The initial set of predicates is empty, but we track control flow (i.e., program counter values) explicitly.

### Iteration 1

**Step**  $1_1$  Since we do not track any predicates, reachability analysis finds that for both threads the entire state space is reachable.

**Step**  $2_1$  To check if the current reach sets are error-free, we check if they contain a state with **Consumer** in location 3 and **Producer** in location 3' (where **data** is accessed). For this, we check the intersection of the regions reachable in locations 3 and 3' for emptiness. Since the respective reachable regions are *true*, we find the intersection nonempty. From the reachability analysis, we have a trace for **Consumer** that leads to 3 with region *true*, and a corresponding trace for **Producer**. We submit both finite traces to counterexample analysis (this routine is discussed in the next section), which reports that the predicates  $flag = 1$  and  $flag = 0$  are important. We reset the havoc assumptions for both threads to *false* and add these two predicates to be tracked.

### Iteration 2

**Step**  $1_2$  Once the reach set for **Consumer** is recomputed using the enlarged set of predicates, we find that the region reachable in locations 3 and 4 is  $flag = 1$ . Hence, **Consumer** reads **data** or modifies **flag** only when  $flag = 1$ . This stems from the assumption that **Producer** never modifies **flag**, which follows from the current havoc assumption *false*. Similarly, in the recomputed reach set for **Producer**, the variables **data** or **flag** are modified only when  $flag = 0$ .

**Step**  $2_2$  We check if the present reach sets contain races. As the intersection of the states when **Consumer** and **Producer** access **data** is  $flag = 1 \wedge flag = 0$ , which is empty, we conclude that there are no races on **data**.

**Step**  $3_2$  We check the soundness of the havoc assumptions by checking if for each thread, the havoc assumptions for **data** and **flag** contain the set of states where the thread modifies the variable. This is not the case, as the havoc assumptions are *false*, but the region where **Consumer** (resp. **Producer**) modifies the variables is  $flag = 1$  (resp.  $flag = 0$ ). Hence we update the havoc assumptions to the new reach sets. In particular, the new havoc assumption of **flag** for **Consumer** (resp. **Producer**) is  $flag = 1$  (resp.  $flag = 0$ ), that is, when analyzing the **Consumer** (resp. **Producer**) we now assume that the environment haves **flag** only when  $flag = 0$  (resp.  $flag = 1$ ).

### Iteration 3

**Step 1<sub>3</sub>** We recompute the reach set of each thread with the new havoc assumptions. When **Consumer** breaks out of the loop at location 1, the state is  $flag = 1$ . Subsequently, the environment cannot havoc **flag**, because the current havoc assumption is  $flag = 0$ . Thus, the state at which **Consumer** accesses **data** has still  $flag = 1$ .

**Step 2<sub>3</sub>** As the reach sets are the same as in iteration 2, there is no race.

**Step 3<sub>3</sub>** This time we find that the havoc assumptions are sound, because they contain the reach sets. Thus we conclude that the current reach sets are over-approximations of the reachable states of the program, and hence there are no races on **data**.  $\square$

## 4 Implementation in BLAST

We have implemented Algorithm TAR in the model checker BLAST [14]. The checker abstracts transition relations and havoc maps using predicate abstraction [12]. The program counter and stack of each thread are kept concrete, but the shared and private state of a thread are abstracted to a boolean combination of a finite set of predicates over program variables.

**Counterexample analysis.** An *atomic region* for a thread consists of the program counter, the stack, and a boolean combination of predicates. For each atomic region  $a$  in the reach set  $R(t_i || g_{3-i})$ , the checker maintains a path  $path(a)$  from an initial region to  $a$ , comprising of  $t_i$  and  $g_{3-i}$  transitions. There is a possible data race on  $x$  if there is a pair of atomic regions  $a \in R(t_i || g_{3-i})$  and  $a' \in R(g_i || t_{3-i})$  such that  $a \subseteq Read_i(x) \cup Write_i(x)$  and  $a' \subseteq Write_{3-i}(x)$  and  $a \cap a' \neq \emptyset$ . If the intersection of the regions  $a$  and  $a'$  is nonempty, then the checker analyzes  $path(a)$  and  $path(a')$ . Our heuristic refinement procedure ignores thread interleavings and symbolically executes  $path(a)$  (resp.  $path(a')$ ) replacing the abstract  $t_i$  (resp.  $t_{3-i}$ ) transitions with concrete  $T_i$  (resp.  $T_{3-i}$ ) transitions. It then uses a theorem prover to check if the conjunction of the resulting symbolic stores is satisfiable. If the conjunction is satisfiable, then it reports a possible race. Note that because the tool ignores thread interleavings, it may return false positives (so far, we have not encountered false positives in practice). If the conjunction is unsatisfiable, then the proof of unsatisfiability is mined for new predicates that are used to refine the abstract transition relations [14].

**Thread symmetry.** If both threads execute the same code, but differ only in the thread identifier, we can optimize the algorithm by computing the reachable set of states for one thread, and obtaining the reachable set of states for the other thread by syntactic renaming. For example, in a variant of Example 1, if both threads run the code of thread 1, we can compute the set of reachable states of thread 2 from those of thread 1 by simultaneously substituting  $m = 2$  for  $m = 1$ ,  $m = 1$  for  $m = 2$ , and  $pc_1$  by  $pc_2$ . This enables our implementation to deal with an unbounded number of threads running the same code.

**Table 2.** Race-checking benchmarks. LOC is lines of code. The number of outer iterations is the number of times the environment assumptions are reset to *false*. The number of inner iterations is the number of times the environment assumptions are updated after the last time they are reset to *false*. Theorem prover queries is the total number of theorem prover queries. Time is the total running time for the tool in seconds on a 700MHz Linux PC with 1GB RAM.

Benchmark	LOC	Iterations		Theorem prover queries	Time (sec)
		outer	inner		
Simple	27	3	97	693	1.208
Simple (buggy)	36	2	131	466	0.220
Producer-Consumer	73	4	505	4349	5.048
Producer-Consumer (buggy)	73	1	149	331	0.403
Time-varying	58	4	612	11219	9.653
Time-varying (buggy)	58	1	61	190	0.259
aironet	1513	3	30955	227735	713.71
packet	4085	3	654	4336	11.610

## 5 Experience

**Locking schemes.** We applied BLAST to a number of small examples that implement different synchronization idioms commonly used in systems code. These include the locking example from Section 2, the producer-consumer synchronization from Section 3, and a time-varying synchronization example from [9]. In the time-varying example, the threads use different locks to access the same shared data at different points in the execution, based on the current state of the program. For each benchmark, we created a second, buggy version by deliberately introducing a bug. In each case, our tool was able to detect the absence or presence of races correctly in a few seconds. Table 2 shows the results of running BLAST on the three benchmarks.

**Device drivers.** We also ran the tool on much larger Linux and Windows NT device drivers. As these drivers can be called by any number of clients concurrently, we are checking for the absence of races on shared variables (typically the state variables maintained by the driver) in an unbounded number of parallel threads. We checked the dispatch routines `readrid` and `writerid` of the `aironet4500` Linux device driver. For this, we modeled the functions `spin_lock` and `spin_unlock` of the Linux kernel manually, as in Example 1, and kernel calls to the driver by an infinite loop that calls the methods `readrid` and `writerid` nondeterministically. Our tool reported that these routines are safe (no data races). We also ran our tool on a network packet driver from the Microsoft Windows DDK. We found two race conditions in this program, which has more than 4,000 lines of C code. The race conditions were benign, because the operations modifying the shared state were atomic. However, further inspection of the traces revealed a real concurrency bug.

## References

1. M. Abadi, L. Lamport. Conjoining specifications. *ACM TOPLAS* 17:507–534, 1995.
2. R. Alur, T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
3. R. Alur, A. Itai, R.P. Kurshan, M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118:142–157, 1995.
4. T. Ball, S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL*, pp. 1–3. ACM, 2002.
5. C. Boyapati, M. Rinard. A parameterized type system for race-free Java programs. In *Proc. OOPSLA*, pp. 56–69, 2001.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, LNCS 1855, pp. 154–169. Springer, 2000.
7. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R.S. Laubach, H. Zheng. BANDERA: extracting finite-state models from Java source code. In *Proc. ICSE*, pp. 439–448. IEEE, 2000.
8. C. Flanagan, S.N. Freund. Detecting race conditions in large programs. In *Proc. PASTE*, pp. 90–96. ACM, 2001.
9. C. Flanagan, S.N. Freund, S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. ESOP*, LNCS 2305, pp. 262–277. Springer, 2002.
10. C. Flanagan, S. Qadeer. Thread-modular model checking. In *Proc. SPIN*, LNCS 2648, pp. 213–224. Springer, 2003.
11. C. Flanagan, S. Qadeer, S.A. Seshia. A modular checker for multithreaded programs. In *Proc. CAV*, LNCS 2404, pp. 180–194. Springer, 2002.
12. S. Graf, H. Säidi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pp. 72–83. Springer, 1997.
13. K. Havelund, T. Pressburger. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer*, 2:72–84, 2000.
14. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy abstraction. In *Proc. POPL*, pp. 58–70. ACM, 2002.
15. G.J. Holzmann. The SPIN model checker. *IEEE TSE*, 23:279–295, 1997.
16. G.J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. SPIN*, LNCS 1885, pp. 131–147. Springer, 2000.
17. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS* 5:596–619, 1983.
18. S. Owicki, D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
19. S. Savage, M. Burrows, C.G. Nelson, P. Sobalvarro, T. Anderson. ERASER: a dynamic data race detector for multithreaded programs. *ACM TOCS*, 15:391–411, 1997.
20. N. Sterling. WARLOCK: a static data race analysis tool. In *Proc. USENIX Technical Conference*, pp. 97–106, 1993.
21. E. Yahav. Verifying safety properties of concurrent Java programs using three-valued logic. In *Proc. POPL*, pp. 27–40. ACM, 2001.

# A Game-Based Framework for CTL Counterexamples and 3-Valued Abstraction-Refinement\*

Sharon Shoham and Orna Grumberg

Computer Science Department, Technion, Haifa, Israel,  
{sharonsh, orna}@cs.technion.ac.il

**Abstract.** This work exploits and extends the game-based framework of CTL model checking for counterexample and incremental abstraction-refinement. We define a game-based CTL model checking for abstract models over the 3-valued semantics, which can be used for verification as well as refutation. The model checking may end with an indefinite result, in which case we suggest a new notion of refinement, which eliminates indefinite results of the model checking. This provides an iterative abstraction-refinement framework. It is enhanced by an *incremental* algorithm, where refinement is applied only where indefinite results exist and definite results from prior iterations are used within the model checking algorithm. We also define the notion of *annotated counterexamples*, which are sufficient and minimal counterexamples for full CTL. We present an algorithm that uses the game board of the model checking game to derive an *annotated counterexample* in case the examined system model refutes the checked formula.

## 1 Introduction

This work exploits and extends the game-based framework [31] of CTL model checking for counterexample and incremental abstraction-refinement.

The first goal of this work is to suggest a game-based new model checking algorithm for the branching-time temporal logic CTL [7] in the context of abstraction. Model checking is a successful approach for verifying whether a system model  $M$  satisfies a specification  $\varphi$ , written as a temporal logic formula. Yet, concrete (regular) models of realistic systems tend to be very large, resulting in the *state explosion problem*. This raises the need for abstraction. Abstraction hides some of the system details, thus resulting in smaller models.

Two types of semantics are available for interpreting CTL formulae over abstract models. The *2-valued* semantics defines a formula  $\varphi$  to be either true or false in an abstract model. True is guaranteed to hold for the concrete model as well, whereas false may be spurious. The *3-valued* semantics [14] introduces a new truth value: the value of a formula on an abstract model may be *indefinite*, which gives no information on its value on the concrete model. On the other hand, both satisfaction and falsification w.r.t the 3-valued semantics hold for the concrete model as well. That is, abstractions over 3-valued semantics are conservative w.r.t. both positive and negative results. They thus give precise results more often both for verification and falsification.

---

\* A fuller version appears in <http://www.cs.technion.ac.il/users/orna/publications.html>

Following the above observation, we define a game-based model checking algorithm for abstract models w.r.t. the 3-valued semantics, where the abstract model can be used for both verification and falsification. However, a third case is now possible: model checking may end with an indefinite answer. This is an indication that our abstraction cannot determine the value of the checked property in the concrete model and therefore needs to be refined. The traditional abstraction-refinement framework [19,6] is designed for 2-valued abstractions, where false may be a false-alarm, thus refinement is aimed at eliminating false results. As such, it is usually based on a counterexample analysis. Unlike this approach, the goal of our refinement is to eliminate indefinite results and turn them into either definite true or definite false.

An advantage of this work lies in the fact that the refinement is then applied only to the indefinite part of the model. Thus, the refined abstract model does not grow unnecessarily. In addition, model checking of the refined model uses definite results from previous runs, resulting in an *incremental* model checking. Our abstraction-refinement process is complete in the sense that for a finite concrete model it will always terminate with a definite “yes” or “no” answer.

The next goal of our work is to use the game-based framework in order to provide counterexamples for full CTL. When model checking a model  $M$  with respect to a property  $\varphi$ , if  $M$  does not satisfy  $\varphi$  then the model checker tries to return a counterexample. Typically, a counterexample is a part of the model that demonstrates the reason for the refutation of  $\varphi$  on  $M$ . Providing counterexamples is an important feature of model checking which helps tremendously in the debugging of the verified system.

Most existing model checking tools return as a counterexample either a finite path (for refuting formulae of the form  $AGp$ ) or a finite path followed by a cycle (for refuting formulae of the form  $AFp^1$ ) [5,7]. Recently, this approach has been extended to provide counterexamples for all formulae of the universal branching-time temporal logic ACTL [9]. In this case the part of the model given as the counterexample has the form of a tree. Other works also extract information from model checking [29,12,25,32]. Yet, it is presented in the form of a temporal proof, rather than a part of the model.

In this work we provide counterexamples for full CTL. As for ACTL, counterexamples are part of the model. However, when CTL is considered, we face existential properties as well. To prove refutation of an existential formula  $E\psi$ , one needs to show an initial state from which *all* paths do not satisfy  $\psi$ . Thus, the structure of the counterexample becomes more complex. Having such a complex counterexample, it might not be easy for the user to analyze it by looking at the subgraph of  $M$  alone. We therefore *annotate* each state on the counterexample with a subformula of  $\varphi$  that is false in that state. The annotating subformulae being false in the respective states, provide the reason for  $\varphi$  to be false in the initial state. Thus, the annotated counterexample gives a convenient tool for debugging. We propose an algorithm that constructs an annotated counterexample and prove that it is sufficient and minimal.

To conclude, the main contributions of this work are:

- A game-based CTL model checking for abstract models over the 3-valued semantics, which can be used for verification as well as refutation.

---

<sup>1</sup>  $AGp$  means “for every path, in every state on the path,  $p$  holds”, whereas  $AFp$  means “along every path there is a state which satisfies  $p$ ”.



- A new notion of refinement, that eliminates indefinite results of the model checking.
- An incremental model checking within the framework of abstraction-refinement.
- A sufficient and minimal counterexample for full CTL.

**Related Work.** Other researchers have suggested abstraction-refinement mechanisms for various branching time temporal logics. In [21] the tearing paradigm is presented as a way to obtain lower and upper approximations of the system. Yet, their technique is restricted to ACTL or ECTL. In [27,28] the full propositional mu-calculus is considered. In their abstraction, the concrete and abstract systems share the same state space. The simplification is based on taking supersets and subsets of a given set with a more compact BDD representation. In [23] full CTL is handled. However, the verified system has to be described as a cartesian product of machines. The initial abstraction considers only machines that directly influence the formula and in each iteration the cone of influence is extended in a BFS manner. [1] handles ACTL and full CTL. Their abstraction collapses all states that satisfy the same subformulae of  $\varphi$  into an abstract state. Thus, computing the abstract model is at least as hard as model checking. Instead, they use partial knowledge on the abstraction function and gain information in each refinement.

Other researchers [14] have suggested to evaluate a property w.r.t the 3-valued semantics by reducing the problem to two 2-valued model checking problems: one for satisfaction and one for refutation. Such a reduction results in the same answer as our algorithm. Yet, it is then not clear how to guide the refinement, in case it is needed.

The game-based approach to model checking, used in this work, is closely related to the Automata-theoretic approach [18], as described in [22]. Thus, our work can also be described in this framework, using alternating automata.

**Organization.** The rest of the paper is organized as follows. In Section 2 we give some background for game-based CTL model checking, abstractions and the 3-valued semantics. Due to technical reasons, we then start with annotated counterexamples. In Section 3 we describe how to construct an annotated counterexample for full CTL and show that it is sufficient and minimal. In Section 4 we extend the game-based model checking to abstract models using the 3-valued semantics. In Section 5 we present our refinement technique, as well as an incremental abstraction-refinement framework.

## 2 Preliminaries

Let  $AP$  be a finite set of atomic propositions. We define the set *Lit* of literals over  $AP$  to be the set  $AP \cup \{\neg p : p \in AP\}$ . We identify  $\neg\neg p$  with  $p$ . In this paper we consider the logic CTL in *negation normal form*, defined as follows:  $\varphi ::= \text{tt} \mid \text{ff} \mid l \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid A\psi \mid E\psi$  where  $l$  ranges over *Lit*, and  $\psi$  is defined by  $\psi ::= X\varphi \mid \varphi U \varphi \mid \varphi V \varphi$ .

The (concrete) semantics of CTL formulae is defined with respect to a *Kripke structure*  $(KS) M = (S, S_0, \rightarrow, L)$ , where  $S$  is a finite set of states,  $S_0 \subseteq S$  is a set of initial states,  $\rightarrow \subseteq S \times S$  is a transition relation, which must be *total* and  $L : S \rightarrow 2^{Lit}$  is a labeling function, such that for every state  $s$  and every  $p \in AP$ ,  $p \in L(s)$  iff  $\neg p \notin L(s)$ .

$[(M, s) \models \varphi] = \text{tt} (= \text{ff})$  means that the CTL formula  $\varphi$  is true (false) in state  $s$  of a KS  $M$ . The formal definition can be found in [7].  $M$  *satisfies*  $\varphi$ , denoted  $[M \models \varphi] = \text{tt}$ , if  $\forall s_0 \in S_0 : [(M, s_0) \models \varphi] = \text{tt}$ . Otherwise,  $M$  *refutes*  $\varphi$ , denoted  $[M \models \varphi] = \text{ff}$ .

**Definition 1.** Given a CTL formula  $\varphi$  of the form  $Q(\varphi_1 U \varphi_2)$  or  $Q(\varphi_1 V \varphi_2)$ , where  $Q \in \{A, E\}$ , its expansion is defined by:

if  $\varphi = Q(\varphi_1 U \varphi_2)$  then  $\text{exp}(\varphi) = \{\varphi, \varphi_2 \vee (\varphi_1 \wedge QX\varphi), \varphi_1 \wedge QX\varphi, QX\varphi\}$   
 if  $\varphi = Q(\varphi_1 V \varphi_2)$  then  $\text{exp}(\varphi) = \{\varphi, \varphi_2 \wedge (\varphi_1 \vee QX\varphi), \varphi_1 \vee QX\varphi, QX\varphi\}$

## 2.1 Game-Based Model Checking Algorithm

Given a (concrete) KS  $M = (S, S_0, \rightarrow, L)$  and a CTL formula  $\varphi$ , the *model checking game* [31,22] of  $M$  and  $\varphi$  is defined as follows. Its board is  $S \times \text{sub}(\varphi)$ , where  $\text{sub}(\varphi)$  is the set of subformulae of  $\varphi$ , defined as usual, except that if  $\varphi = A(\varphi_1 U \varphi_2)$ ,  $E(\varphi_1 U \varphi_2)$ ,  $A(\varphi_1 V \varphi_2)$  or  $E(\varphi_1 V \varphi_2)$  then  $\text{sub}(\varphi) = \text{exp}(\varphi) \cup \text{sub}(\varphi_1) \cup \text{sub}(\varphi_2)$ .

The model checking game is played by two players,  $\forall$ belard, the refuter, and  $\exists$ loise, the prover. A *play* is a (possibly infinite) sequence  $C_0 \rightarrow_{p_0} C_1 \rightarrow_{p_1} C_2 \rightarrow_{p_2} \dots$  of configurations, where  $C_0 \in S_0 \times \{\varphi\}$ ,  $C_i \in S \times \text{sub}(\varphi)$  and  $p_i \in \{\forall, \exists\}$ . The subformula in  $C_i$  determines which player  $p_i$  makes the next move.

### Possible Moves at Each Step

1.  $C_i = (s, \text{ff})$ ,  $C_i = (s, \text{tt})$ , or  $C_i = (s, l)$  where  $l \in \text{Lit}$ : the play is finished. Such configurations are called *terminal configurations*.
2.  $C_i = (s, AX\varphi)$ :  $\forall$ belard chooses a transition  $s \rightarrow s'$  and  $C_{i+1} = (s', \varphi)$ .
3.  $C_i = (s, EX\varphi)$ :  $\exists$ loise chooses a transition  $s \rightarrow s'$  and  $C_{i+1} = (s', \varphi)$ .
4.  $C_i = (s, \varphi_1 \wedge \varphi_2)$ :  $\forall$ belard chooses  $j \in \{1, 2\}$  and  $C_{i+1} = (s, \varphi_j)$ .
5.  $C_i = (s, \varphi_1 \vee \varphi_2)$ :  $\exists$ loise chooses  $j \in \{1, 2\}$  and  $C_{i+1} = (s, \varphi_j)$ .
6.  $C_i = (s, Q(\varphi_1 U \varphi_2))$ ,  $Q \in \{A, E\}$ :  $C_{i+1} = (s, \varphi_2 \vee (\varphi_1 \wedge QXQ(\varphi_1 U \varphi_2)))$ .
7.  $C_i = (s, Q(\varphi_1 V \varphi_2))$ ,  $Q \in \{A, E\}$ :  $C_{i+1} = (s, \varphi_2 \wedge (\varphi_1 \vee QXQ(\varphi_1 V \varphi_2)))$ .

In configurations 6-7 the move is deterministic, thus any player can make the move. A play is *maximal* iff it is infinite or ends in a terminal configuration. In [31] it is shown that a play is infinite iff exactly one subformula of the form  $AU$ ,  $EU$ ,  $AV$  or  $EV$  occurs in it infinitely often. Such a subformula is called a *witness*.

**Winning Criteria.**  $\forall$ belard wins the play iff (1) the play is finite and ends in a terminal configuration of the form  $C_i = (s, \text{ff})$ , or  $C_i = (s, l)$ , where  $l \notin L(s)$ , or (2) the play is infinite and the witness is  $AU$  or  $EU$ .  $\exists$ loise wins otherwise.

The model checking *game* consists of all the possible plays. A *winning strategy* is a set of rules for a player, telling him how to move in the current configuration and allowing him to win every play if he plays by the rules. All possible plays of a game are captured in the *game-graph*. It is the graph whose nodes are the elements (configurations) of the game board and whose edges are the possible moves of the players.

The model checking algorithm for the evaluation of  $[M \models \varphi]$  consists of two parts. First, it constructs (part of) the *game-graph*. The evaluation of the truth value of  $\varphi$  in  $M$  is then done by coloring the game-graph.

**Game-Graph Construction and Its Properties.** The subgraph of the game-graph that is reachable from the initial configurations  $S_0 \times \{\varphi\}$  is constructed in a BFS or DFS manner. It is denoted  $G_{M \times \varphi} = (N, E)$ , where  $N \subseteq S \times \text{sub}(\varphi)$ . The nodes (configurations) of  $G_{M \times \varphi}$  can be classified into three types. Terminal configurations are *leaves*

in the game-graph. Nodes whose subformulae are of the form  $\varphi_1 \wedge \varphi_2$  or  $AX\varphi_1$  are  $\wedge$ -nodes. Nodes whose subformulae are of the form  $\varphi_1 \vee \varphi_2$  or  $EX\varphi_1$  are  $\vee$ -nodes. Nodes whose subformulae are  $AU$ ,  $EU$ ,  $AV$ ,  $EV$  can be considered either  $\vee$ -nodes or  $\wedge$ -nodes. Sometimes we further distinguish between nodes whose subformulae are of the form  $AX\varphi$  ( $EX\varphi$ ) and other  $\wedge$ -nodes ( $\vee$ -nodes), by referring to them as  $AX$ -nodes ( $EX$ -nodes). The edges in  $G_{M \times \varphi}$  are divided to *progress edges*, that originate in  $AX$ -nodes or  $EX$ -nodes and reflect transitions of the KS, and *auxiliary edges*, which are the rest. Each non-trivial *strongly connected component* (SCC) in  $G_{M \times \varphi}$ , i.e. an SCC with one edge at least, contains exactly one witness and is classified as an  $AU$ ,  $AV$ ,  $EU$ , or  $EV$  SCC, based on its witness.

**Coloring Algorithm.** The following *Coloring Algorithm* [3] labels each node in  $G_{M \times \varphi}$  by  $T$  or  $F$ , depending on whether  $\exists$ loise or  $\forall$ belard has a winning strategy.  $G_{M \times \varphi}$  is partitioned into its *Maximal Strongly Connected Components* (MSCCs), denoted  $Q_i$ 's, and an order  $\leq$  is determined on them, such that an edge  $(n, n')$ , where  $n \in Q_i$  and  $n' \in Q_j$ , exists in  $G_{M \times \varphi}$  only if  $Q_j \leq Q_i$ . Such an order exists because the MSCCs form a *directed acyclic graph* (DAG). It can be extended to a total order  $\leq$  arbitrarily.

The coloring algorithm processes the  $Q_i$ 's according to  $\leq$ , bottom-up. Let  $Q_i$  be the smallest MSCC w.r.t  $\leq$  that is not yet fully colored. Every outgoing edge of  $Q_i$  leads to a colored node or remains in the same set,  $Q_i$ . The nodes of  $Q_i$  are colored as follows.

1. Terminal nodes are colored by  $T$  if  $\exists$ loise wins in them, and by  $F$  otherwise.
2. An  $\vee$ -node ( $\wedge$ -node) is colored by  $T$  ( $F$ ) if it has a son that is colored by  $T$  ( $F$ ), and by  $F$  ( $T$ ) if all its sons are colored by  $F$  ( $T$ ).
3. All the nodes in  $Q_i$  that remain uncolored, after the propagation of these rules, are colored according to the witness in  $Q_i$ . They are colored by  $F$  if the witness is of the form  $AU$  or  $EU$ , and are colored by  $T$  if the witness is of the form  $AV$  or  $EV$ .

The result of the coloring algorithm is a *coloring function*  $\chi : N \rightarrow \{T, F\}$ .

**Theorem 1.** [31] *Let  $M$  be a KS,  $\varphi$  a CTL formula and  $(s, \varphi_1) \in G_{M \times \varphi}$ . Then:*

*$[(M, s) \models \varphi_1] = tt$  ( $ff$ )  $\Leftrightarrow \exists$ loise ( $\forall$ belard) has a winning strategy for the game starting at  $(s, \varphi_1) \Leftrightarrow (s, \varphi_1)$  is colored by  $T$  ( $F$ ).*

## 2.2 Abstraction

Abstract models preserving CTL need to have two transition relations [20,11]. This is achieved by using *Kripke Modal Transition Systems* [17,13].

**Definition 2.** A Kripke Modal Transition System (*KMTS*) is a tuple  $M = (S, S_0, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L)$ , where  $S$  is a finite set of states,  $S_0 \subseteq S$  is a set of initial states,  $\xrightarrow{\text{must}} \subseteq S \times S$  and  $\xrightarrow{\text{may}} \subseteq S \times S$  are transition relations such that  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$ , and  $L : S \rightarrow 2^{Lit}$  is a labeling function, s.t. for each state  $s$  and  $p \in AP$ , at most one of  $p$  and  $\neg p$  is in  $L(s)$ .

We consider abstractions that collapse sets of concrete states into single abstract states. Such abstractions can be described in the framework of *Abstract Interpretation* [24,11]. Let  $M_C = (S_C, S_{0C}, \rightarrow, L_C)$  be a (concrete) KS. Let  $(S_A, \sqsubseteq)$  be a poset of abstract states and  $(\gamma : S_A \rightarrow 2^{S_C}, \alpha : 2^{S_C} \rightarrow S_A)$  a *Galois connection* [10,24]

from  $(2^{S_C}, \subseteq)$  to  $(S_A, \sqsubseteq)$ .  $\gamma$  is the *concretization function* that maps each abstract state to the set of concrete states that it represents.  $\alpha$  is the *abstraction function* that maps each set of concrete states to the abstract state that represents it.

An abstract model  $M_A$  can then be defined as follows. The set of initial abstract states  $S_{0A}$  is defined such that  $s_{0a} \in S_{0A}$  iff there exists  $s_{0c} \in S_{0C}$  for which  $s_{0c} \in \gamma(s_{0a})$ . An abstract state  $s_a$  is labeled by  $l \in Lit$  only if all the concrete states that it represents are labeled by  $l$ . Thus, it is possible that neither  $p$  nor  $\neg p$  are in  $L_A(s_a)$ . The *may*-transitions are computed s.t. they represent (at least) every concrete transition: if  $\exists s_c \in \gamma(s_a)$  and  $\exists s'_c \in \gamma(s'_a)$  s.t.  $s_c \rightarrow s'_c$ , then  $s_a \xrightarrow{\text{may}} s'_a$ . The *must*-transitions represent concrete transitions that are common to all the concrete states represented by the origin abstract state:  $s_a \xrightarrow{\text{must}} s'_a$  only if  $\forall s_c \in \gamma(s_a) \exists s'_c \in \gamma(s'_a)$  s.t.  $s_c \rightarrow s'_c$ . Other constructions of abstract models, based on Galois connections, can be found in [11,15].

The relation  $H \in S_C \times S_A$ , which is defined by  $(s_c, s_a) \in H$  iff  $s_c \in \gamma(s_a)$ , then forms a *mixed simulation* [11,13] from  $M_C$  to the resulting abstract model  $M_A$ .

[17] defines the 3-valued semantics of CTL over KMTSSs, denoted  $[(M, s) \models^3 \varphi]$ , preserving both satisfaction (tt) and refutation (ff) from the abstract model to the concrete one. However, a new truth value,  $\perp$ , is introduced, meaning that the truth value over the concrete model is not known and can be either tt or ff.

**Theorem 2.** [13] *Let  $H \subseteq S_C \times S_A$  be a mixed simulation relation from a KS  $M_C$  to a KMTS  $M_A$ . Then for every  $(s_c, s_a) \in H$  and every CTL formula  $\varphi$ :*

$$[(M_A, s_a) \models^3 \varphi] = tt \text{ (ff)} \Rightarrow [(M_C, s_c) \models \varphi] = tt \text{ (ff)}$$

### 3 Using Games to Produce Annotated Counterexamples

In this section we describe how to construct an *annotated counterexample* from the coloring of a game-graph for  $M$  and  $\varphi$  in case  $M$  does not satisfy  $\varphi$ .

First, the coloring algorithm is changed to identify and remember the *cause* of the coloring of an  $\wedge$ -node  $n$  that is colored by  $F$ . If  $n$  was colored by its sons, then  $cause(n)$  is the son that was the first to be colored by  $F$ . If  $n$  was colored due to a witness, then  $cause(n)$  is chosen to be one of its sons which resides on the same SCC and was colored by witness as well. There must exist such a son, otherwise  $n$  would be colored by its sons. Note that  $cause(n)$  depends on the execution of the coloring algorithm.

Given a game-graph  $G_{M \times \varphi}$ , for a KS  $M$  and a CTL formula  $\varphi$ , and given its coloring  $\chi$  and an initial node  $n_0 = (s_0, \varphi)$  s.t.  $\chi(n_0) = F$ , the following algorithm finds an *annotated counterexample*, denoted  $C$ , which is a subgraph of  $G_{M \times \varphi}$  colored by  $F$ .

**Algorithm** ComputeCounter

Initially:  $new = \{(s_0, \varphi)\}$ ,  $C = \emptyset$ .

while  $new \neq \emptyset$

$n = \text{remove}(new)$

- if  $n$  was already handled or if  $n$  is a terminal node - continue.

- if  $n$  is an  $\vee$ -node, then for each son  $n'$  of  $n$  add  $n'$  to  $new$  and  $(n, n')$  to  $C$ .

- if  $n$  is an  $\wedge$ -node, then add  $cause(n)$  to  $new$  and  $(n, cause(n))$  to  $C$ .

**Complexity.** Algorithm ComputeCounter has a linear running time (in the worst case) w.r.t the size of the game-graph  $G_{M \times \varphi}$ . The latter is bounded by  $O(|M| \cdot |\varphi|)$ .

Note, that for the correctness of  $C$  it is *mandatory* to choose for an  $\wedge$ -node the son that caused the coloring of the node, and not any son that was colored by  $F$ .

**Properties of the Computed Annotated Counterexample.**  $C$  is a subgraph of  $G_{M \times \varphi}$  s.t. for each node  $n \in C$ ,  $\chi(n) = F$ . It can be viewed as the part of the winning strategy of the refuter that is sufficient to guarantee his victory. We formalize and prove this notion in the next section. Intuitively, it is indeed a counterexample in the sense that it points out the reasons for  $\varphi$ 's refutation on the model. Each node in  $C$  is marked by a state  $s$  and a subformula  $\varphi_1$ , s.t.  $\chi((s, \varphi_1)) = F$ , thus by Theorem 1,  $[s \models \varphi_1] = \text{ff}$ . The edges point out the reason (cause) for the refutation of a certain subformula in a certain state: the refutation in an  $\wedge$ -node is shown by refutation in one of its sons, whereas the refutation in an  $\vee$ -node is shown by all its sons. Another important property is:

**Lemma 1.**  *$C$  contains non-trivial SCCs iff at least one of the nodes in the SCC was colored due to a witness. We conclude that non-trivial SCCs in  $C$  are AU- or EU-SCCs.*

The property of  $C$  described in Lemma 1 implies that any non-trivial SCC that appears in the annotated counterexample indicates a refutation of the  $U$  operator, which results, at least partly, from an infinite path, where weak until is satisfied, but not strong until.

### 3.1 The Annotated Counterexample Is Sufficient and Minimal

In this section we first informally describe our requirements of a counterexample. We then formalize them for annotated counterexamples and show that they are fulfilled by the result of `ComputeCounter`. Generally speaking, for a sub-model to be a counterexample, it is expected to: (1) falsify the given formula; (2) hold “enough” information to explain why the model refutes the formula; and (3) be minimal in the sense that removing any state or transition will not maintain 1 and 2. To formalize the second requirement w.r.t an annotated counterexample, we need the following definitions.

**Definition 3.** *Let  $G = (N, E)$  be a game-graph and let  $A$  be a subgraph of  $G$ . The partial coloring algorithm of  $G$  w.r.t  $A$  works as follows. It is given an initial coloring function  $\chi_I : N \setminus A \rightarrow \{T, F\}$  and computes a coloring function for  $G$ . The algorithm is identical to the (original) coloring algorithm, except for the addition of a new rule:*

- A node  $n \in N \setminus A$  is colored by  $\chi_I(n)$  and its color is not changed.

*Any result of the partial coloring algorithm of  $G$  with respect to  $A$  is called a partial coloring function of  $G$  with respect to  $A$ , denoted  $\overline{\chi} : N \rightarrow \{T, F\}$ .*

As opposed to the usual coloring algorithm that has only one possible result, the partial coloring algorithm has several possible results, depending on the initial coloring function  $\chi_I$ . Each one of them is considered a partial coloring function of  $G$  w.r.t  $A$ . By definition, the usual coloring algorithm is a partial coloring algorithm of  $G$  w.r.t  $G$ .

**Definition 4.** *Let  $G$  be a game-graph and let  $\chi$  be the result of the coloring algorithm on  $G$ . A subgraph  $A$  of  $G$  is independent of  $G$  if for each  $\overline{\chi}$  that is a partial coloring function of  $G$  with respect to  $A$ , and for each  $n \in A$ , we have that  $\chi(n) = \overline{\chi}(n)$ .*

Basically, a subgraph  $A$  is independent of  $G$  if its coloring is *absolute* in the sense that all of its completions to the full game-graph do not change the color of any node in  $A$ .

We can now formalize the notion of an annotated counterexample.

**Definition 5.** Let  $G$  be a game-graph, and let  $\chi$  be its coloring function, such that  $\chi(n_0) = F$  for some initial node  $n_0$ . A subgraph  $\tilde{C}$  of  $G$  containing  $n_0$  is an annotated counterexample if it satisfies the following conditions. (1) For each node  $n \in \tilde{C}$ ,  $\chi(n) = F$ ; (2)  $\tilde{C}$  is independent of  $G$ ; and (3)  $\tilde{C}$  is minimal.

The first two requirements in Definition 5 imply that  $\tilde{C}$  is *sufficient* for explaining why  $n_0$  is colored  $F$ : First it guarantees that all the nodes in  $\tilde{C}$  are colored  $F$ . In addition, since  $\tilde{C}$  is independent of  $G$ , we can conclude that regardless of the other nodes in  $G$ , all the nodes in  $\tilde{C}$ , and in particular  $n_0$ , will be colored  $F$ . Thus, it also explains why the model falsifies the formula. The third condition shows that  $\tilde{C}$  is also “necessary”.

We now show that the result of `ComputeCounter`, denoted  $C$ , is indeed an annotated counterexample. The first requirement is obviously fulfilled, as described earlier. The following theorem states that  $C$  satisfies the other two conditions as well.

**Theorem 3.**  $C$  is independent of  $G$ . Moreover,  $C$  is minimal in the sense that removing a node or an edge will result in a subgraph that is not independent of  $G$ .

The correctness of the first part of Theorem 3 strongly depends on the choice of  $\text{cause}(n)$  as the son of an  $\wedge$ -node in the algorithm `ComputeCounter`.

## 4 Game-Based Model Checking on Abstract Models

In this section we suggest a generalization of the game-based model checking algorithm for evaluating a CTL formula  $\varphi$  over a KMTS  $M$  w.r.t the 3-valued semantics.

We start with the description of the 3-valued game. The main difference arises from the fact that KMTSs have two types of transitions. Since the transitions of the model are considered only in configurations with subformulae of the form  $AX\varphi_1$  or  $EX\varphi_1$ , these are the only cases where the rules of the play need to be changed. Intuitively, in order to be able to both prove and refute each subformula, the game needs to allow the players to use both may and must transitions in such configurations. The reason is that for example, truth of a formula  $AX\varphi_1$  should be checked upon may-transitions, but its falseness should be checked upon must-transitions.

### New Moves of the Game

2. if  $C_i = (s, AX\varphi)$ , then  $\forall$ belard chooses a transition  $s \xrightarrow{\text{must}} s'$  (for refutation) or  $s \xrightarrow{\text{may}} s'$  (for satisfaction), and  $C_{i+1} = (s', \varphi)$ .
3. if  $C_i = (s, EX\varphi)$ , then  $\exists$ loise chooses a transition  $s \xrightarrow{\text{must}} s'$  (for satisfaction) or  $s \xrightarrow{\text{may}} s'$  (for refutation), and  $C_{i+1} = (s', \varphi)$ .

Intuitively, the players use must-transitions in order to win, while they use may transitions in order to prevent the other player from winning. As a result it is possible that none of the players wins the play, i.e. the play ends with a tie. As before, a *maximal* play is infinite if and only if exactly one *witness*, which is either an  $AU, EU, AV$  or  $EV$ -formula, appears in it infinitely often. However, the winning rules become more complicated. A player can only win the play if he or she are “consistent” in their moves:

**Definition 6.** A player is said to play consistently if in each configuration where he proceeds over the transitions of the model, his move is based on a  $\xrightarrow{\text{must}}$  transition.

### New Winning Criteria

- $\forall$ belard wins a play iff he plays consistently and in addition one of the following holds: (1) The play is finite and ends in a configuration  $C_i = (s, \text{ff})$  or  $(s, l)$ , where  $\neg l \in L(s)$ ; or (2) The play is infinite and the witness is of the form  $AU$  or  $EU$ .
- $\exists$ loise wins a play iff she plays consistently and in addition one of the following holds: (1) the play is finite and ends in configuration  $C_i = (s, \text{tt})$  or  $(s, l)$ , where  $l \in L(s)$ ; or (2) the play is infinite and the witness is of the form  $AV$  or  $EV$ .
- Otherwise, the play ends with a tie.

**Theorem 4.** Let  $M$  be a KMTS and  $\varphi$  a CTL formula. Then, for each  $s \in S$ :

1.  $[(M, s) \stackrel{3}{\models} \varphi] = \text{tt}$  iff  $\exists$ loise has a winning strategy for the game starting at  $(s, \varphi)$ .
2.  $[(M, s) \stackrel{3}{\models} \varphi] = \text{ff}$  iff  $\forall$ belard has a winning strategy for the game starting at  $(s, \varphi)$ .
3.  $[(M, s) \stackrel{3}{\models} \varphi] = \perp$  iff none of them has a winning strategy for the game from  $(s, \varphi)$ .

In order to use this correspondence for model checking, we generalize the game-based model checking algorithm. The (3-valued) game-graph, denoted  $G_{M \times \varphi}$ , is constructed as in the “concrete” case. Its nodes, denoted  $N$ , are again classified as  $\wedge$ -nodes,  $\vee$ -nodes,  $AX$ -nodes or  $EX$ -nodes. Similarly, the edges are classified as *progress* edges or *auxiliary* edges. But now, we distinguish between two types of progress edges: Edges that are based on must-transitions are referred to as *must-edges*. Edges that are based on may-transitions are referred to as *may-edges*. A node  $n'$  is a *may-son* (*must-son*) of the node  $n$  if there exists a may-edge (must-edge) from  $n$  to  $n'$ . An SCC in the game-graph is a *may-SCC* (*must-SCC*) if all its progress edges are may-edges (must-edges).

The coloring algorithm of the 3-valued game-graph needs to be adapted as well. First, a new color, denoted  $?$ , is introduced for configurations in which none of the players has a winning strategy. Second, the partition to  $Q_i$ ’s that is based on MSCCs is now based on may-MSCCs (note that  $\xrightarrow{\text{must}} \subseteq \xrightarrow{\text{may}}$ ).

### The (3-Valued) Coloring Algorithm

**Partition and Order.**  $G_{M \times \varphi}$  is partitioned into its may-MSCCs, denoted  $Q'_i$ ’s. A (total) order  $\leq$  is determined on them in the same way as for the concrete case.

**Coloring.** As before, the coloring algorithm processes the  $Q_i$ ’s bottom-up. Let  $Q_i$  be the smallest set w.r.t  $\leq$  that is not yet fully colored. Its nodes are colored in two phases.

1. *Sons-coloring phase.* Apply the following rules to  $Q_i$  until none is applicable.
  - A terminal node is colored by  $T$  if  $\exists$ loise wins in it, by  $F$  if  $\forall$ belard wins in it, and by  $?$  otherwise.
  - An  $AX$ -node ( $EX$ -node) is colored by:
    - $T$  ( $F$ ) if all its may-sons are colored  $T$  ( $F$ ).
    - $F$  ( $T$ ) if it has a must-son that is colored  $F$  ( $T$ ).
    - $?$  if all its must sons are colored  $T$  ( $F$ ) or  $?$  and it has a may-son that is colored  $F$  ( $T$ ) or  $?$ .

- An  $\wedge$ -node ( $\vee$ -node), other than  $AX$ -node ( $EX$ -node), is colored by:
  - $T$  ( $F$ ) if both its sons are colored  $T$  ( $F$ ).
  - $F$  ( $T$ ) if it has a son that is colored  $F$  ( $T$ ).
  - $?$  if it has a son that is colored  $?$  and the other is colored  $?$  or  $T$  ( $F$ ).
- 2. *Witness-coloring phase.* If after the propagation of the rules of phase 1 there are still uncolored nodes in  $Q_i$ , then  $Q_i$  must be a non-trivial may-MSCC that has exactly one witness. Its uncolored nodes are colored according to the witness, as follows.
  - The witness is of the form  $A(\varphi_1 U \varphi_2)$  or  $E(\varphi_1 U \varphi_2)$ :
    - (a) Repeatedly color  $?$  each node in  $Q_i$  satisfying one of the following.
      - An  $\wedge$ -node ( $AX$ -node) that all its (must) sons are colored  $T$  or  $?$ .
      - An  $\vee$ -node ( $EX$ -node) that has a (may) son that is colored  $T$  or  $?$ .
    - (b) Color the remaining nodes in  $Q_i$  by  $F$ .
  - The witness is of the form  $A(\varphi_1 V \varphi_2)$  or  $E(\varphi_1 V \varphi_2)$ :
    - (a) Repeatedly color  $?$  each node in  $Q_i$  satisfying one of the following.
      - An  $\wedge$ -node ( $AX$ -node) that has a (may) son that is colored  $F$  or  $?$ .
      - An  $\vee$ -node ( $EX$ -node) that all its (must) sons are colored  $F$  or  $?$ .
    - (b) Color the remaining nodes in  $Q_i$  by  $T$ .

The result of the coloring algorithm is a 3-valued coloring function  $\chi : N \rightarrow \{T, F, ?\}$ . Note that a node is colored  $?$  only if there is evidence that it cannot be colored otherwise.

**Theorem 5.** *Let  $G_{M \times \varphi}$  be a 3-valued game-graph, then for each  $n \in G_{M \times \varphi}$ :*

1.  $\chi(n) = T$  iff  $\exists$ loise has a winning strategy for the game starting at  $n$ .
2.  $\chi(n) = F$  iff  $\forall$ belard has a winning strategy for the game starting at  $n$ .
3.  $\chi(n) = ?$  iff none of the players has a winning strategy for the game starting at  $n$ .

The correctness of the coloring algorithm is strongly based on the property that when phase 2b is applied, the uncolored nodes that are colored in it form non-trivial SCCs. In case of an  $AU$ -witness, these are must-SCCs, and indeed in this case loops can only be used for refutation, thus to identify “real” loops, must-edges are needed. On the other hand, in case of an  $AV$ -witness, loops can contribute to satisfaction, and satisfaction of universal properties should be examined upon may-transitions, and indeed for such a witness, we get uncolored may-SCCs. Similarly, for an  $EU$  witness, we get may-SCCs, whereas for an  $EV$  witness, must-SCCs are formed.

**Implementation Issues and Complexity.** The coloring algorithm can be implemented in linear running time w.r.t the size of  $G_{M \times \varphi}$ , using a variation of an AND/OR graph, similarly to the algorithm described in [18] for checking nonemptiness of the language of a simple weak alternating word automaton. Thus, its running time is  $O(|M| \cdot |\varphi|)$ .

As a conclusion of Theorem 4 and Theorem 5, we get the following theorem.

**Theorem 6.** *Let  $M$  be a KMTS,  $\varphi$  a CTL formula and  $(s, \varphi_1) \in G_{M \times \varphi}$ . Then:*

$$[(M, s) \models^3 \varphi_1] = tt, ff \text{ or } \perp \Leftrightarrow (s, \varphi_1) \text{ is colored by } T, F \text{ or } ? \text{ respectively.}$$

Given the colored game-graph, if all the initial nodes are colored  $T$ , or if at least one of them is colored  $F$ , then by Theorem 6 and Theorem 2, there is a definite answer as for the satisfaction of  $\varphi$  in the *concrete* model. This is because there exists a mixed simulation from the concrete to the abstract model. Furthermore, if the result is ff, a *concrete* annotated counterexample can be produced, using an extension of ComputeCounter.



## 5 Refinement

In this section we show how to exploit the abstract game-graph in order to refine the abstract model in case model checking resulted in an indefinite answer. When the result is  $\perp$ , there is no reason to assume either one of the definite answers tt or ff. Thus, we would like to base the refinement not on a counterexample as in [19,6,2,8,4], but on the point(s) that are responsible for the uncertainty. The goal of the refinement is to discard these points, in the hope of getting a definite result on the refined abstraction.

Let  $M_C = (S_C, S_{0C}, \rightarrow, L_C)$  be a concrete KS and let  $M_A = (S_A, S_{0A}, \xrightarrow{\text{must}}, \xrightarrow{\text{may}}, L_A)$  be an abstract KMTS. Let  $\gamma : S_A \rightarrow 2^{S_C}$  be the concretization function. Given the abstract 3-valued game-graph  $G$ , based on  $M_A$ , and its coloring function  $\chi : N \rightarrow \{T, F, ?\}$ , such that  $\chi(n_0) = ?$  for some initial node  $n_0$ , we use the information gained by the coloring algorithm of  $G$  in order to refine the abstraction.

Refinement is done by splitting abstract states according to criteria obtained from *failure nodes*. A node is a *failure node* if it is colored by  $?$ , whereas none of its sons was colored by  $?$  at the time it got colored by the algorithm. Such a node is a failure node in the sense that it can be seen as the point where the loss of information occurred. Note, that a terminal node that is colored by  $?$  is also considered a failure node. The coloring algorithm is adapted to remember failure nodes. In addition, for each node  $n$  that is colored by  $?$ , but is *not* a failure node, the coloring algorithm remembers a son that was already colored  $?$  by the time  $n$  was colored, denoted  $\text{cont}(n)$ .

**Searching for a Failure Node.** A *failure node* is found by a DFS-like greedy algorithm, starting from  $n_0$ : As long as the current node,  $n$ , is not a failure node, the algorithm proceeds to  $\text{cont}(n)$ . It ends and returns  $n$  when a failure node  $n$  is reached.

**Lemma 2.** *A failure node is either (1) a terminal node; (2) an AX-node (EX-node) that has a may-son colored by F (T); or (3) an AX-node (EX-node) that was colored during phase 2a based on an AU (EV) witness, and has a may-son colored by ?.*

**Failure Analysis.** Based on the failure node  $n$ , the refinement is reduced to the problem of separating sets of (concrete) states, which can be solved by known techniques, depending on the abstraction used (e.g. [8,6]).  $n$  provides the criterion for the separation:

1.  $n = (s_a, l)$  is a terminal node. The reason for its indefinite color is that  $s_a$  represents both concrete states that are labeled by  $l$  and by  $\neg l$ . This is avoided by separating  $\gamma(s_a)$  to two sets  $\{s_c \in \gamma(s_a) : l \in L_C(s_c)\}$  and  $\{s_c \in \gamma(s_a) : \neg l \in L_C(s_c)\}$ .
2.  $n = (s_a, AX\varphi_1)$  or  $(s_a, EX\varphi_1)$  with a may-son colored  $F$  or  $T$  resp. Let  $K$  stand for  $F$  or  $T$ . We define  $\text{sons}_K = \bigcup \{\gamma(s'_a) : (s'_a, \varphi_1) \text{ is a may son of } n \text{ colored } K\}$  and  $\text{conc}_K = \gamma(s_a) \cap \{s_c \in S_C : \exists s'_c \in \text{sons}_K, s_c \rightarrow s'_c\}$ . For the  $AX\varphi_1$  case,  $K = F$  and  $\text{conc}_K$  is the set of all concrete states, represented by  $s_a$ , that definitely refute  $AX\varphi_1$ . For the  $EX\varphi_1$  case,  $K = T$  and  $\text{conc}_K$  is the set of all concrete states, represented by  $s_a$ , that definitely satisfy  $EX\varphi_1$ . In both cases, our goal is to separate the sets  $\text{conc}_K$  and  $\gamma(s_a) \setminus \text{conc}_K$ .
3.  $n = (s_a, AX\varphi_1)$  or  $(s_a, EX\varphi_1)$  was colored during phase 2a based on an  $AU$  or an  $EV$  witness resp., and has a may-son  $n' = (s'_a, \varphi_1)$  colored by  $?$ . Let  $\text{conc}_? = \gamma(s_a) \cap \{s_c \in S_C : \exists s'_c \in \gamma(s'_a), s_c \rightarrow s'_c\}$  be the set of all concrete states, represented by  $s_a$ , that have a son represented by  $s'_a$ . Our goal is to separate the sets  $\text{conc}_?$  and  $\gamma(s_a) \setminus \text{conc}_?$ .

It is possible that one of the sets obtained during the failure analysis is empty and provides no criterion for the split. Yet, this is informative as well. As an example, consider case 2, where the failure node  $n$  is an  $AX$ -node. If  $\text{conc}_F = \gamma(s_a)$ , then every state represented by  $s_a$  has a refuting son. Thus,  $n$  can be colored  $F$  instead of  $?$ . If  $\text{conc}_F = \emptyset$ , then none of the concrete states in  $\gamma(s_a)$  has a transition to a concrete state represented by the  $F$ -colored may-sons of  $n$ . Thus, the may-edges from  $n$  to such sons can be removed. Either way,  $G$  can be recolored starting from the  $Q_i$  containing  $n$ .

The purpose of the split derived from cases 1-2 is to conclude definite results about (at least part) of the new abstract states obtained by the split of the failure node. These results can be used by the incremental algorithm, suggested below. As for case 3, we know that by the time the failure node  $n$  got colored, its may-son  $n'$  that is colored by  $?$  was not yet colored (otherwise  $n$  would not be a failure node). By the description of the algorithm, if  $n'$  was a must-son of  $n$ , then as long as it was uncolored,  $n$  would remain uncolored too and would eventually be colored in phase 2b by a definite color. Thus, our goal in this case is to obtain a must edge between (parts of)  $n$  and  $n'$ .

**Theorem 7.** *For finite concrete models, iterating the abstraction-refinement process is guaranteed to terminate with a definite answer.*

## 5.1 Incremental Abstraction-Refinement Framework

We refine abstract models by splitting their states. The criterion for the refinement is decided locally, based on one node, but has a global effect. Yet, there is no reason to split states for which the model checking results are definite. The game-based model checking provides a convenient framework to use previous results, leading to an *incremental* model checking based on iterative abstraction-refinement, where each iteration consists of abstraction, model checking and refinement. After each iteration, we now remember the (abstract) nodes colored by definite colors, as well as nodes for which a definite color was discovered during failure analysis. When a refined game-graph is constructed, it is pruned in nodes that are *sub-nodes* of nodes remembered from previous iterations. A node  $(s_a, \varphi)$  is a *sub-node* of  $(s'_a, \varphi')$  if  $\varphi = \varphi'$  and the concrete states represented by  $s_a$  are a subset of those represented by  $s'_a$ . Thus, only the reachable subgraph that was previously colored  $?$  is refined. The coloring algorithm considers the nodes where the game-graph was pruned as leaves and colors them by their previous colors.

Note that for many abstractions, checking if a node is a sub-node of another is simple. For example, in the framework of predicate abstraction [16,30,26,15], this means that the abstract states “agree” on all the predicates that exist before the refinement.

## References

1. A. Asteroth, C. Baier, and U. Assmann. Model checking with formula-dependent abstract models. In *Computer Aided Verification*, volume 2102 of *LNCS*, pages 155–168, 2001.
2. Sharon Barner, Daniel Geist, and Anna Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In *Computer Aided Verification*, 2002.

3. Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free mu-calculus. In *SPIN'02*. Springer-Verlag Inc., 2002.
4. P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D.Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD)*, November 2002.
5. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC'95*. IEEE Computer Society Press, 1995.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, LNCS, Chicago, USA, July 2000.
7. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT press, December 1999.
8. E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer-Aided Verification*, July 2002.
9. E.M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS)*, July 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *popl4*, pages 238–252, 1977.
11. Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), March 1997.
12. D.Peled, A.Pnueli, and L.Zuck. From falsification to verification. In *FSTTCS*, 2001.
13. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Computer-Aided Verification*, volume 2404 of LNCS, pages 137–150, July 2002.
14. P. Godefroid and R. Jagadeesan. On the expressiveness of 3-valued models. In *Proc. of VMCAI*, volume 2575 of LNCS, pages 206–222. Springer-Verlag, January 2003.
15. Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of CONCUR'01*, 2001.
16. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
17. Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. *LNCS*, 2028:155–169, 2001.
18. Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM (JACM)*, 47(2):312–360, 2000.
19. R.P. Kurshan. *Computer-Aided-Verification of Coordinating Processes*. Princeton University Press, 1994.
20. K.G. Larsen and B. Thomsen. A modal process logic. In *LICS*, pages 203–210, 1988.
21. Woohyuk Lee, Abelardo Pardo, Jae-Young Jang, Gary D. Hachtel, and Fabio Somenzi. Tearing based automatic abstraction for CTL model checking. In *ICCAD*, pages 76–81, 1996.
22. Martin Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In *Conf. on Logic for Programming and Automated Reasoning (LPAR)*, 1999.
23. Jorn Lind-Nielsen and Henrik Reif Andersen. Stepwise CTL model checking of state/event systems. In *Computer Aided Verification*, pages 316–327, 1999.
24. 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*, 1995.
25. Kedar S. Namjoshi. Certifying model checkers. In *CAV*, volume 2102 of LNCS, 2001.
26. Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, volume 1855 of LNCS, pages 435–449. Springer, 2000.
27. Abelardo Pardo and Gary D. Hachtel. Automatic abstraction techniques for propositional mu-calculus model checking. In *Computer Aided Verification*, pages 12–23, 1997.
28. Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Design Automation Conference (DAC)*, pages 457–462, 1998.
29. Doron Peled and Lenore Zuck. From model checking to a temporal proof. In *SPIN*, 2001.
30. H. Saidi and N. Shankar. Abstract and model check while you prove. In *CAV*, 1999.
31. Colin Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.
32. Li Tan and Rance Cleaveland. Evidence-based model checking. In *CAV*, 2002.

# Abstraction for Branching Time Properties

Kedar S. Namjoshi

Bell Labs, Lucent Technologies  
kedar@research.bell-labs.com

**Abstract.** Effective program abstraction is needed to successfully apply model checking in practice. This paper studies the question of constructing abstractions that preserve branching time properties. The key challenge is to simultaneously preserve the existential and universal aspects of a property, *without* relying on bisimulation. To achieve this, our method abstracts an *alternating transition system* (ATS) formed by the product of a program with an alternating tree automaton for a property. The AND-OR distinction in the ATS is used to guide the abstraction, weakening the transition relation at AND states, and strengthening it at OR states. We show *semantic completeness*: i.e., whenever a program satisfies a property, this can be shown using a *finite-state* abstract ATS produced by the method. To achieve completeness, the method requires *choice predicates* that help resolve nondeterminism at OR states, and rank functions that help preserve progress properties. Specializing this result to predicate abstraction, we obtain exact characterizations of the types of properties provable with these methods.

## 1 Introduction

It is generally accepted that effective, automated, program abstraction is central to the successful application of model checking techniques. Methods include the use of transformations (e.g., [11,21]), and predicate abstraction (e.g., [19,4,29,3,33,20]). Most current methods use abstractions that preserve universal temporal properties, such as those expressible in linear temporal logic (LTL) and ACTL\*. In several settings, there is a need for methods that preserve the full range of branching time properties, including mixed existential and universal ones – for instance, to analyze programs with unresolved non-determinism, or to analyze process-environment interaction [1].

The key challenge is to simultaneously preserve both the universal and the existential aspects of a branching time property during abstraction. Methods for universal properties rely on establishing a simulation relation [26] from the concrete to an abstract program. To preserve all branching time properties in a similar manner requires a bisimulation between the two programs, which is too restrictive. We propose a method that abstracts not the program, but rather the product *alternating transition system* (ATS) that is formed from the program

and an alternating tree automaton for the property. The ATS can be abstracted through local transformations, weakening the transition relation at AND states, and strengthening it at OR states, in a manner similar to that of [14].

As the verification problem is undecidable in general even for invariance properties, no algorithm exists that can always construct a finite-state abstract program precise enough to prove that a concrete program satisfies a property. Hence, we look for *semantic completeness*: if a program satisfies a property, can this fact be shown using a finite state abstract program constructed by the method, ignoring computability issues? (In practice, the computability question corresponds to using a powerful enough decision procedure.) Uribe [32] and Kesten and Pnueli [22] showed that simulation-based abstraction must be augmented with fairness to be complete for LTL progress properties, such as termination. The fairness constraints serve to abstractly represent such termination requirements.

This completeness result does not, however, apply to all universal properties; for instance, to logics such as ACTL. Technically, the problem is that disjunction of temporal state formulas, as in  $AX(p) \vee AX(\neg p)$ , cannot be represented in LTL. Thus, for branching time properties, we have the distinction of AND vs. OR branching in addition to the invariance-progress distinction that was considered earlier. This distinction is important: we show that to achieve completeness, one requires *choice predicates* at OR states, which provide hints for the resolution of the OR nondeterminism, in addition to the rank predicates needed for abstracting liveness properties. As in [32,22], our completeness result links the *deductive provability* of a property on a program — using a proof system designed in [27] — to the construction of a finite abstract ATS. The analysis reveals that abstraction without any augmentation can be used only to verify those properties that have proofs with *uniform* OR choice (a precise definition is given later), and *bounded* progress measures, showing why both types of augmentation are needed to verify arbitrary properties.

We examine the consequences of these results for *predicate abstraction*, which defines the abstract state space in terms of boolean variables that correspond to concrete program predicates. Automatically discovering relevant predicates is a key problem, for which several heuristics have been proposed (e.g., [29,8]). Based on our completeness results, we can give an exact characterization of the properties provable using such discovery methods.

To summarize, the main contribution of this paper is a new abstraction method for branching time properties, which does not rely on bisimulation. This is the first such method for branching time properties that is known to be complete. As corollaries, we derive several completeness results for predicate discovery procedures. In addition, a key intermediate theorem shows how to construct a deductive proof of correctness on the concrete state space from a feasible abstract ATS. Such proof construction is of independent interest [28].

## 2 Preliminaries

In this section, we recall the definition of alternating tree automata (ATA), and the product construction used in model checking. For the rest of the paper, we fix an action set  $\Sigma$  and a set of atomic propositions,  $AP$ .

An *labeled transition system* (LTS)  $M$  over  $\Sigma$  and  $AP$  is defined by a tuple  $(S, I, R, L)$ , where  $S$  is a set of states,  $I$  is the subset of initial states,  $R \subseteq S \times \Sigma \times S$  is a transition relation, and  $L : S \rightarrow 2^{AP}$  is a labeling of states with atomic propositions. We assume that the relation  $R$  is total over  $\Sigma$ ; i.e., for each  $a$  in  $\Sigma$ , every state has an  $a$ -successor.

An *alternating tree automaton* (ATA) over  $\Sigma$  and  $AP$  is given by a tuple  $(Q \cup \{tt, ff\}, \hat{q}, \delta, F)$ , where  $Q$  is a finite set of states,  $\hat{q}$  in  $Q$  is the initial state,  $\delta$  is a transition relation, and  $F = (F_0, \dots, F_{2n})$  is a partition of  $Q$ , called the *parity acceptance condition* [17]. A sequence of automaton states is accepted if *the least* index  $i$  such that a state from  $F_i$  occurs infinitely often on the sequence is *even*. The transition relation  $\delta$  maps an automaton state, and a predicate on  $AP$  to one of:  $ff$  (an error state);  $tt$  (an accept state);  $q_1 \wedge q_2$  (forking off automaton copies in state  $q_1$  and  $q_2$ );  $q_1 \vee q_2$  (choosing to proceed in either state  $q_1$  or state  $q_2$ );  $\langle a \rangle q_1$  (continuing in  $q_1$  for some  $a$ -successor);  $[a]q_1$  (continuing in  $q_1$  for every  $a$ -successor).

An LTS  $M = (S, I, R, L)$  *satisfies* a property given by an ATA  $A = (Q, \hat{q}, \delta, F)$  iff player I has a winning strategy in an infinite, two player game [17]. In this game, a configuration is a pair of a computation tree node of  $M$  labeled by a state  $s$ , and an automaton state  $q$ . A configuration labeled  $(s, q)$  is a win for player I if  $\delta(q, L(s)) = tt$ ; it is a loss if  $\delta(q, L(s)) = ff$ . For other values of  $\delta$ , player I picks the next move iff  $\delta(q, L(s))$  is either  $\langle a \rangle q_1$  or  $q_1 \vee q_2$ . Player I picks an  $a$ -successor for  $s$  for  $\langle a \rangle q_1$ , or the choice of disjunct. Similarly, player II picks an  $a$ -successor for  $s$  for  $[a]q_1$ , or the choice of conjunct for  $q_1 \wedge q_2$ . A play of the game is a win for player I iff it either ends in a win for I, or it is infinite and the sequence of automaton states on it satisfies  $F$ . A *strategy* is a function mapping a partial play to the next move; given strategies for players I and II, one can generate the possible plays. Finally, the LTS satisfies the automaton property iff player I has a *winning* strategy (one for which every generated play is a win for player I) for the game played on the computation tree of the LTS from the initial configuration labeled  $(\hat{s}, \hat{q})$ .

Every closed formula of the  $\mu$ -calculus [24] can be translated in linear time to an equivalent ATA [17]. The transition relation of the automaton has the following simple form: each state has a single transition for the input predicate *true*, except for a transition to  $tt$  on predicate  $l$ ; in which case, there is another transition to  $ff$  on  $\neg l$ . In the rest of the paper, we work with such simple automata.

An *alternating transition system* (ATS) over a set  $\Gamma$  of state labels is defined by a tuple  $(S, I, R, L)$ , where  $S$ , the set of states, is partitioned into AND and OR subsets,  $I$  is a set of initial states,  $R \subseteq S \times S$  is the transition relation, and  $L : S \rightarrow \Gamma$  is the state labeling.

The *model checking* problem is to determine whether  $M$  satisfies  $A$  at all initial states, and is written as  $M \models A$ . This can be determined using a product ATS,  $M \times A$ , defined by  $(S, I, R, L)$ , where  $S = S_M \times Q_A$ ,  $I = I_M \times \{\hat{q}_A\}$ , and  $L : (s, q) \rightarrow q$ .  $R((s, q), (s', q'))$  holds based on the value of  $\delta_A(q, l)$ , as follows: (i)  $\text{ff}, \text{tt}$ :  $q' = \delta_A(q, l) \wedge s' = s \wedge l(s)$ , (ii)  $q_1 \wedge q_2, q_1 \vee q_2$ :  $q' \in \{q_1, q_2\} \wedge s' = s$ , as  $l \equiv \text{true}$ , and (iii)  $\langle a \rangle q_1, [a] q_1$ :  $q' = q_1 \wedge R_M(s, a, s')$ , as  $l \equiv \text{true}$ . A state is an OR state if  $\delta_A(q, l)$  is either  $q_1 \vee q_2$  or  $\langle a \rangle q_1$ , and an AND state otherwise. In [16], it is shown that  $M$  satisfies  $A$  iff player I has a winning strategy in the game graph defined by  $M \times A$ , where player I makes choices at OR states, and player II at AND states, and that this can be determined by model checking.

### 3 The Abstraction Method

We define our abstraction method for the product alternating transition system (ATS). Soundness is shown by constructing a valid concrete proof of correctness given the feasibility of the abstract ATS. This construction also indicates why augmentation with choice predicates and rank functions is needed in general. In the following, let  $M = (S, I, R, L)$  be an LTS over  $\Sigma$  and  $AP$ , and let  $A = (Q, \hat{q}, \delta, F)$ , where  $F = (F_0, \dots, F_{2n})$  (for some  $n$ ), be an ATA over  $\Sigma$  and  $AP$ . Let  $M \times A$  be the product ATS, constructed as shown earlier.

The basic abstraction idea is simple: we are given an abstract domain  $\overline{S}$ , and a set of left-total *abstraction relations*  $\{\xi_q \mid q \in Q\}$ , where each  $\xi_q \subseteq S \times \overline{S}$ . We also define  $\xi_{\text{tt}}$  and  $\xi_{\text{ff}}$  to be  $S \times \overline{S}$ , and say that  $(s, q)$  is related to  $(t, q)$  iff  $s\xi_q t$  holds. We abstract the ATS  $M \times A$  by *weakening* its transition relation at AND states (thus allowing “more” transitions), and *strengthening* it at OR states (thus “eliminating” some transitions). The result is an abstract ATS, denoted by  $\overline{M \times A}$ . The abstract ATS is given by  $(S', I', R', L')$ .

- The abstract set of states,  $S' = \overline{S} \times Q$ . The abstract OR states are those where  $\delta(q, \text{true})$  has the form  $q_1 \vee q_2$  or  $\langle a \rangle q_1$ , all others are AND states. Thus, related concrete and abstract states have the same AND/OR tag.
- The abstract state  $(t, \hat{q})$  is in  $I'$  if there exists  $s \in I$  for which  $s\xi_{\hat{q}} t$ .
- The abstract transition relation,  $R'$ , is given by:
  - For an abstract AND state  $(t, q)$ , the transition  $((t, q), (t', q'))$  is in  $R'$  if there exists a concrete state  $(s, q)$  and a successor  $(s', q')$  that are related to  $(t, q), (t', q')$  respectively.

$$R'((t, q), (t', q')) \Leftarrow (\exists s : s\xi_q t : (\exists s' : s'\xi_{q'} t' : R((s, q), (s', q'))))$$

- For an abstract OR state  $(t, q)$ , the transition  $((t, q), (t', q'))$  is in  $R'$  *only if* for every  $(s, q)$  which is related to  $(t, q)$ , there exists a successor  $(s', q')$  which is related to  $(t', q')$ .

$$R'((t, q), (t', q')) \Rightarrow (\forall s : s\xi_q t : (\exists s' : s'\xi_{q'} t' : R((s, q), (s', q'))))$$

- The abstract labeling function  $L'$  maps a state  $(t, q)$  to  $q$ .

**Consistency:** For each  $q \in Q$ ,  $[\phi_q \Rightarrow (\exists k : (\rho_q = k))]$  ( $\rho_q$  is defined for every state in  $\phi_q$ )

**Initiality:**  $[I \Rightarrow \phi_{\hat{q}}]$  (every initial state satisfies the initial invariant)

**Invariance and Progress:** For each  $q \in Q$ , and predicate  $l$  over  $AP$ , based on the form of  $\delta(q, l)$ , check the following.

- $tt$ : there is nothing to check.
- $ff$ :  $[\phi_q \Rightarrow \neg l]$  holds,
- $q_1 \wedge q_2$ :  $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q_1} \wedge (\rho_{q_1} \triangleleft_q k)) \wedge (\phi_{q_2} \wedge (\rho_{q_2} \triangleleft_q k))]$
- $q_1 \vee q_2$ :  $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow (\phi_{q_1} \wedge (\rho_{q_1} \triangleleft_q k)) \vee (\phi_{q_2} \wedge (\rho_{q_2} \triangleleft_q k))]$
- $\langle a \rangle q_1$ :  $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow \langle a \rangle (\phi_{q_1} \wedge (\rho_{q_1} \triangleleft_q k))]$
- $[a] q_1$ :  $[\phi_q \wedge l \wedge (\rho_q = k) \Rightarrow [a] (\phi_{q_1} \wedge (\rho_{q_1} \triangleleft_q k))]$

**Fig. 1.** Deductive Proof System for Automaton Properties

*Precise Abstraction:* Notice that the method allows some flexibility in the definition of  $R'$ . If  $R'$  is defined in a way that the implications become equivalences, we say that  $R'$  is *precise* (precision of abstractions is studied in depth in [10,14]). This flexibility can be exploited in practice by doing approximate but faster calculations of a less precise  $R'$ . Precise abstractions are needed for completeness, though, as is shown later. Note also that the abstract ATS can be constructed by symbolic calculations, thus avoiding the explicit construction of  $M \times A$ .

**Theorem 0 (Soundness)** *For any LTS  $M$  and alternating tree automaton  $A$ , let  $\overline{M \times A}$  be defined by the abstraction method, based on a set of abstraction relations  $\{\xi_q\}$ . Then, if  $\overline{M \times A}$  is feasible, so is  $M \times A$ .*

One can prove this theorem by showing that the relation  $\alpha$  given by:  $(s, q)\alpha(t, q')$  iff  $q = q'$  and  $s\xi_q t$  is an alternating refinement relation [1] which preserves the labeling (i.e., the automaton component). Therefore, any winning strategy for player I in  $\overline{M \times A}$ , induces (through the refinement) a winning strategy on  $M \times A$ . However, we use a different argument, showing how to construct a deductive proof that  $M$  satisfies  $A$ , given that  $\overline{M \times A}$  is feasible. This construction provides information useful for the completeness proof.

*Deductive Proofs:* A deductive proof system (from [27]) for proving that  $M$  satisfies  $A$  is shown in Fig. 1. One needs: (i) for each automaton state  $q$ , an *invariance* predicate,  $\phi_q$ , which is a subset of  $S$ , (ii) non-empty, well ordered sets  $W_1, \dots, W_n$  with associated partial orders  $\preceq_1, \dots, \preceq_n$ . Let  $W = W_1 \times \dots \times W_n$ , and let  $\preceq$  be the lexicographic well order defined on  $W$  from  $\{\preceq_i\}$ , (iii) for each automaton state  $q$ , a partial *rank function*  $\rho_q : S \rightarrow W$ . Let  $\prec^i$  be the restriction of  $\prec$  to the first  $i$  components. For an automaton state  $q$ , the rank change predicate  $(a \triangleleft_q b)$  holds either if  $q$  belongs to an odd indexed  $F_{2i-1}$  and  $a \prec^i b$ , or if  $q$  is in an even indexed  $F_{2i}$  and  $a \preceq^i b$  (this odd/even distinction is clearly related to the parity condition). A proof is valid if it meets the conditions given the figure.



**Theorem 1** [27] *For program  $M$  and automaton  $A$ ,  $M \models A$  iff there is a valid deductive proof that  $M$  satisfies  $A$ .*

**Theorem 2 (Proof Construction)** *If  $\overline{M \times A}$  is feasible, there is a valid deductive proof that  $M$  satisfies  $A$ .*

**Proof Sketch.** Let  $\mathcal{A}$  be the set of states that are wins for player I in  $\overline{M \times A}$ . The proof construction is similar to that in [27] so, for lack of space, we present only a short sketch. The proof  $\Pi$  is defined by  $(\phi, \rho, W)$ , where: (i)  $\phi_q(s)$  holds iff there exists  $t$  such that  $s\xi_q t$  and  $(t, q)$  is in  $\mathcal{A}$ , (ii)  $\rho_q(s)$  is the minimum of the signatures of states  $t$  such that  $s\xi_q t$  and  $(t, q) \in \mathcal{A}$ , and (iii)  $W$  is the domain of the signatures. The signature [31] of a state in  $\mathcal{A}$  is an  $n$ -tuple of ordinals which, roughly, measures the progress made toward satisfying  $A$ 's acceptance condition. For example, if  $A$  has a Büchi acceptance condition, it is the distance in  $M \times A$  to an accepting state.

We consider in detail only the case where  $\delta(q, \text{true}) = \langle a \rangle q_1$ . If  $\phi_q(s)$  and  $(\rho_q(s) = k)$  holds for any state  $s$  of  $M$  and any  $k \in W$ , from the definitions, there is  $t$  such that  $s\xi_q t$ ,  $(t, q) \in \mathcal{A}$ , and  $t$  has signature  $k$ . As  $(t, q)$  is an OR state, it has a successor,  $(t', q_1)$ , in  $\mathcal{A}$ . By the  $\forall\exists$  abstraction for OR states,  $(s, q)$  must have a successor  $(s', q_1)$  such that  $s'\xi_{q_1} t'$ . Thus,  $s'$  is an  $a$ -successor of  $s$  in  $M$ , and  $\phi_{q_1}(s')$  holds. By a property of signatures [31], the signature of  $(t', q_1)$  is  $k'$ , where  $k' \triangleleft_q k$ . By definition,  $\rho_{q_1}(s') \preceq k'$ , so that  $\rho_{q_1}(s') \triangleleft_q k$ .  $\square$

**Proof of Theorem 0:** The soundness theorem follows from the combination of Theorems 1 and 2. If  $\overline{M \times A}$  is feasible, by Theorem 2, there is a valid concrete proof that  $M$  satisfies  $A$ . Applying Theorem 1, it follows that  $M$  satisfies  $A$ .  $\square$

Theorem 2 gives valuable information about the constructed proof  $\Pi$  if  $\overline{M \times A}$  is finite-state:

- **(Uniform OR Choice)** By the  $\forall\exists$  nature of the abstraction at OR states, for  $q$  such that  $\delta(q, \text{true}) = q_1 \vee q_2$ , if  $(t, q)$  has a transition to  $(t', q_1)$ , then for every  $s$  such that  $s\xi_q t$  holds, there is a transition from  $(s, q)$  to  $(s, q_1)$ . A similar observation holds for the  $\langle a \rangle$  case.
- **(Bounded Progress)** As the abstract ATS is finite-state, the rank domain  $W$  is a finite set. Thus,  $\Pi$  shows that  $M$  satisfies  $A$  using only bounded progress measures.

These restrictions mean that the abstraction procedure, although sound, cannot be complete. To illustrate this, consider showing the property  $\text{EF}(x \geq 0)$  (i.e., there exists a future where  $x \geq 0$ ) for the following program  $M$ .

```
var x: integer; initially true
actions (a) x := x-1 (b) x := x+1
```

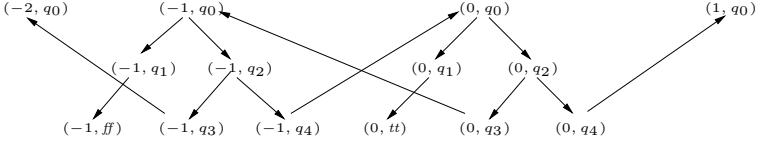
The property is true at every initial state, but showing this requires a proof with unbounded progress measure (the measure  $\rho(x) = -x$ , if  $x < 0$ , else 0.). The automaton  $A$  for the property is defined below, and a fragment of the product ATS is shown in Fig. 2.

States:  $\{q_0, q_1, q_2, q_3, q_4\}$ ; Initial state:  $q_0$

Transitions:  $\delta(q_0, \text{true}) = q_1 \vee q_2$ ;  $\delta(q_1, x \geq 0) = tt$ ;  $\delta(q_1, x < 0) = ff$ ;

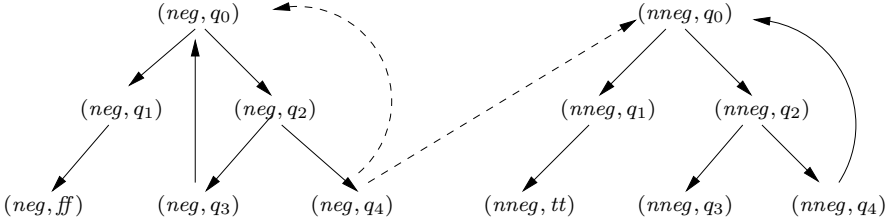
$\delta(q_2, \text{true}) = q_3 \vee q_4$ ;  $\delta(q_3, \text{true}) = \langle a \rangle q_0$ ;  $\delta(q_4, \text{true}) = \langle b \rangle q_0$

Parity condition:  $(\{q_1\}, \{q_0, q_2, q_3, q_4\})$



**Fig. 2.** A portion of  $M \times A$

Now consider the abstract ATS in Fig. 3. Here, *neg* stands for  $\{x \mid (x < 0)\}$  and *nneg* for  $\{x \mid (x \geq 0)\}$ . Solid lines indicate transitions that are in the precise abstraction. However, these transitions, in themselves, are not sufficient for feasibility since there is no way to pick transitions so that it is possible to satisfy the acceptance condition from the abstract state  $(\text{neg}, q_0)$ . We would like, in particular, the dashed transitions from state  $(\text{neg}, q_4)$  to exist in the abstraction, but these are ruled out by the strong  $\forall\exists$  nature of the abstraction at OR states. Clearly, it is not possible for all states where  $x < 0$  to have a *b*-transition to a state where  $x < 0$ , and similarly, it is not possible for all such states to have a *b*-transition to a state where  $x \geq 0$ . On the other hand, the  $\forall\exists$  abstraction is needed for soundness at OR states. Moreover, no finite refinement of the *neg* state will resolve this problem.



**Fig. 3.** A Possible Abstraction

*Choice Predicates:* A way out of this dilemma is given by the introduction of *choice predicates*. The essential idea is to weaken the  $\forall$  quantification at an abstract OR state  $(t, q)$  in the  $\forall\exists$  abstraction to apply only to a subset of the states in  $\xi_q^{-1}(t)$ . These subsets are supplied to the abstraction procedure through a partial function  $\epsilon$ , defined for a subset of OR-states, called the *choice states* in the sequel. At each choice state  $(t, q)$ , the function supplies, for a possible

transition to a state  $(t', q')$ , a predicate  $\epsilon((t, q), (t', q'))$  (note that the transition is possible if either  $\delta(q, \text{true}) = q_1 \vee q_2$ , and  $q' \in \{q_1, q_2\}$ , or  $\delta(q, \text{true}) = \langle a \rangle q_1$  and  $q' = q_1$ ). An abstract transition  $((t, q), (t', q'))$  from a choice state  $(t, q)$  is computed by restricting the  $\forall$  quantification in the  $\forall\exists$  abstraction to states satisfying its choice predicate. The union of choice predicates for all transitions in  $R'$  from  $(t, q)$  should be a superset of  $\xi_q^{-1}(t)$ .

At  $(\text{neg}, q_4)$ , we let the choice predicate for the transition to  $(\text{neg}, q_0)$  be  $(x < -1)$ , and the predicate for the transition to  $(\text{nneg}, q_0)$  be  $(x = -1)$ . This adds back the dashed transitions in Fig. 3. However, it also creates a different problem. The transition from  $(\text{neg}, q_4)$  to  $(\text{neg}, q_0)$  introduces an infinite loop, which does not exist in the original ATS, since this transition increments the value of  $x$ . To solve this difficulty, we use rank functions in a manner similar to that in [32,22].

We suppose that a set of rank function  $\{\eta_q \mid q \in Q\}$  is supplied, which map states in  $S$  to a well-founded set with the structure specified in the proof system. We add to each abstract transition a label ('good' or 'bad') indicating whether the rank given by  $\eta$  changes in a way appropriate to the change of automaton state along the transition. The final method, for the supplied abstraction relations  $\{\xi_{qj}\}$ , rank functions  $\{\eta_q\}$ , and the choice function  $\epsilon$  is given below. The abstract ATS is given by  $(S', I', R', L')$ , where  $S'$ ,  $I'$ , and  $L'$  are defined as before. The definition of the abstract transition relation,  $R'$ , is modified to the following, where  $g$  is the label on the abstract transition.

- For an abstract AND state  $(t, q)$ ,

$$R'((t, q), g, (t', q')) \Leftarrow (\exists s : s\xi_q t : (\exists s' : s'\xi_{q'} t' : R((s, q), (s', q')) \wedge g \equiv \eta_{q'}(s') \triangleleft_q \eta_q(s)))$$

- For an abstract choice state  $(t, q)$ ,

$$R'((t, q), g, (t', q')) \Rightarrow \epsilon((t, q), (t', q')) \neq \emptyset \wedge (\forall s : s\xi_q t \wedge s \in \epsilon((t, q), (t', q')) : (\exists s' : s'\xi_{q'} t' : R((s, q), (s', q')) \wedge g \equiv \eta_{q'}(s') \triangleleft_q \eta_q(s)))$$

- The transitions from abstract OR states are as for choice states with choice predicate  $\epsilon \equiv \text{true}$ .

Taking, in addition to the choice augmentation discussed above, the rank functions where for  $s$  such that  $x(s) < 0$ ,  $\eta_{q_0}(s) = -3 * x(s)$ ,  $\eta_{q_1}(s) = -3 * x(s) - 1$ ,  $\eta_{q_2} = -3 * x(s) - 2$ , and for  $x(s) \geq 0$ , all values are 0, and applying the augmented abstraction procedure, we obtain the abstract ATS shown in Fig. 4, where  $\eta$ -good transitions are labeled with  $*$ .

We only consider abstract ATS's where at a choice state  $(t, q)$ , the union of the choice predicates on its successors together form a superset of  $\xi_q^{-1}(t)$ . We define a game on such abstract ATS which is identical to the game defined in Section 2, except that: (a) player II has the choice of successor at every choice state, and (b) all infinite plays satisfy *either* the parity acceptance condition of the

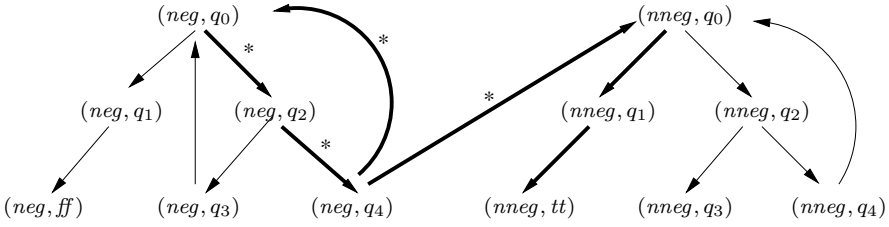


Fig. 4. Abstract ATS after Augmented Abstraction

automaton or, from some point on, contain only  $\eta$ -good transitions. We say that an abstract ATS is *subtly feasible* if player I has a winning strategy in the new game. The bold transitions in Fig. 4 indicate such a winning strategy.

**Theorem 3 (Soundness of Augmented Abstraction)** *For any LTS  $M$  and alternating tree automaton  $A$ , let  $\overline{M} \times \overline{A}$  be defined by the augmented abstraction procedure above, based on a set of abstraction relations  $\{\xi_q\}$ , choice predicate  $\epsilon$ , and rank functions  $\{\eta_q\}$ . If  $\overline{M} \times \overline{A}$  is subtly feasible, then  $M \times A$  is feasible.*

**Proof.** As  $\overline{M} \times \overline{A}$  is subtly feasible, player I has a winning strategy on it. Let  $\mathcal{A}$  be the set of winning states for player I. We use this strategy to provide a winning strategy for the player on  $M \times A$ . Inductively, we assume that for a state  $(s, q)$  on a play following this strategy, there is an abstract state  $(t, q)$  in  $\mathcal{A}$  such that  $s\xi_q t$  holds. This is true for the initial states of  $M \times A$  as every initial state of the abstract ATS is in  $\mathcal{A}$ .

Consider any state  $(s, q)$  on a play, and let  $(t, q)$  be its corresponding abstract state. If  $(s, q)$  is an AND-state, let  $(s', q')$  be any successor chosen by player II. By the definition of the abstract ATS, there is a successor  $(t', q')$  of  $(t, q)$  that matches it. Now suppose that  $(s, q)$  is an OR state, and that  $(t, q)$  is a choice state. Let  $(t', q')$  be a successor of  $(t, q)$  (which is in  $\mathcal{A}$  by the new game rules) such that  $s \in \epsilon((t, q), (t', q'))$ . Such a successor must exist because the choice predicates at  $(t, q)$  cover  $\xi_q^{-1}(t)$ . The  $\forall\exists$  abstraction implies that  $(s, q)$  has a successor  $(s', q')$  such that  $s'\xi_{q'} t'$  holds. Player I picks this successor. A similar argument applies to ordinary OR states.

Any constructed play  $\pi$  has a corresponding play  $\pi'$  in the winning game on the abstract ATS, which agrees with it on the sequence of automaton states. So if  $\pi$  is maximal and finite, it must end in a state with label  $tt$ . If  $\pi$  is infinite, we show by contradiction that it must satisfy  $F$ . If not, then  $\pi'$  does not do so either, and the new acceptance condition implies that it eventually consists of only  $\eta$ -good transitions. Thus, from some point on, the least  $F$ -index in  $\pi$  is *odd* (say  $2k - 1$ ), and all of its transitions correspond to  $\eta$ -good transitions in  $\pi'$ . At each such concrete transition, the  $\eta$ -goodness condition, with the definition of  $\triangleleft$ , ensure that  $\eta$ , restricted to the first  $k$  components, does not increase, and strictly decreases infinitely often. This contradicts the well-foundedness of the rank domain.  $\square$

## 4 Completeness

**Theorem 4 (Completeness)** *If  $M$  satisfies an ATA property  $A$ , there is an augmented abstraction  $\bar{M} \times \bar{A}$  that is subtly feasible.*

**Proof.** Since  $M$  satisfies  $A$ , by Theorem 1, there is a valid proof  $\Pi = (\phi, \rho, W)$  of this fact. The constructed abstraction follows the proof very closely, as is also the case for the constructions in [32,22] for LTL.

Let the abstract domain  $\bar{S}$  be the set  $\{a_q \mid q \in Q\} \cup \{a_{tt}, a_{ff}\} \cup \{\perp\}$ . Let the abstraction relations for  $q \in Q$  be:  $s \xi_q t$  iff  $\phi_q(s) \wedge t = a_q$  or  $\neg \phi_q(s) \wedge t = \perp$  holds. The state  $\perp$  is an unreachable state used to ensure that each  $\xi_q$  is left-total. The rank functions used for abstraction are  $\{\rho_q\}$  from the proof. The choice predicate is defined only for those abstract OR states  $(a_q, q)$  where  $\delta(q, \text{true}) = q_1 \vee q_2$ :  $\epsilon((a_q, q), (a_{q'}, q'))$  is the set  $\{s \mid \phi_{q'}(s) \wedge \rho_{q'}(s) \triangleleft_q \rho_q(s)\}$ , and is empty for other possible successor states. We claim that the *precise* abstract program constructed with these choices is subtly feasible. Inductively, the winning strategy ensures that the only reachable configurations are those of the form  $(a_q, q)$ , where  $q$  is in  $Q \cup \{tt\}$ , and  $\phi_q$  is non-empty. This is true of the only initial state,  $(a_{\hat{q}}, \hat{q})$ .

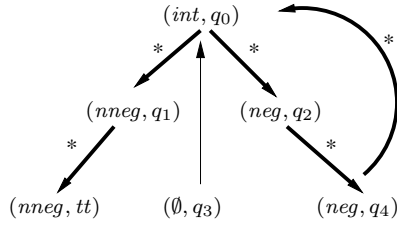
Suppose  $(a_q, q)$  is reached according to some partial play. Since  $\phi_q$  is non-empty, there is a related concrete state  $(s, q)$ . If  $\delta(q, \text{true}) = q_1 \wedge q_2$ , then by the proof,  $s$  satisfies  $\phi_{q_1}$  and  $\phi_{q_2}$ . Thus,  $(a_q, q)$  has  $(a_{q_1}, q_1)$  and  $(a_{q_2}, q_2)$  as successor states. As the abstraction is precise, these are the only possible successors. Now suppose that  $\delta(q, \text{true}) = [a]q'$ . Then, by the proof, for every  $a$ -successor  $(s', q')$  of  $(s, q)$  in  $M \times A$ ,  $\phi_{q'}(s')$  holds, and there is at least one such successor, so that  $(a_q, q)$  has  $(a_{q'}, q')$  as its only successor. If  $\delta(q, l) = ff$  (so that  $\delta(q, \neg l) = tt$ ), by the proof,  $[\phi_q \Rightarrow \neg l]$ , so  $(a_q, q)$  can only have the successor  $(a_{tt}, tt)$ . Suppose that  $(a_q, q)$  is a choice state, so that  $\delta(q, \text{true}) = q_1 \vee q_2$ . By the proof,  $\xi_q^{-1}(a_q) = \phi_q$  is covered by the two choices, so that every state in  $\phi_q$  satisfies some choice, and that  $(a_{q_1}, q_1)$  and  $(a_{q_2}, q_2)$  are the only possible successors of this abstract state. Otherwise,  $(a_q, q)$  is an OR state and  $\delta(q, \text{true}) = \langle a \rangle q'$ . By the proof, every state  $s$  satisfying  $\phi_q$  has an  $a$ -successor satisfying  $\phi_{q'}$ . Hence,  $(a_q, q)$  has a transition to  $(a_{q'}, q')$ , which is picked by player I.

It follows from the proof that every abstract transition is  $\rho$ -good. Thus, every infinite play either satisfies the parity acceptance condition or has only  $\rho$ -good transitions, and every maximal finite path in  $\mathcal{A}$  must end with automaton component  $tt$ . Hence, the abstract ATS is subtly feasible.  $\square$

The abstract ATS constructed in this manner for the example program is given in Fig. 5, with the winning strategy outlined in bold (*int* stands for the set of all integers). This uses the same rank functions as for Fig. 4, and invariants defined by the abstract state component.

### 4.1 Predicate Abstraction

Predicate abstraction[19] defines the abstract state space in terms of boolean variables corresponding to concrete program predicates. Computing a relevant



**Fig. 5.** Abstract ATS derived from the completeness proof

set of predicates is impossible in general [20]; however, there are several heuristics for *predicate discovery*. The general predicate discovery scheme [29,8] starts with a set  $\mathcal{P}_0$  of predicates from the correctness property, and iteratively computes  $\mathcal{P}_{i+1}$  by adding the predicates in the weakest precondition *wp* [15] of  $\mathcal{P}_i$  to  $\mathcal{P}_i$ . Let the limit of this procedure be denoted by the set  $\mathcal{P}^*$ .

From a modification of the main completeness theorem, we can derive the reverse direction of the following theorem. Note that the forward direction holds from the first soundness theorem. Thus, we obtain an exact characterization of the completeness of predicate discovery methods.

**Theorem 5 (*Relative Completeness of Predicate Discovery*)** *For an LTS  $M$  and ATA  $A$ , predicate discovery coupled with abstraction produces a feasible result iff there is a proof that  $M$  satisfies  $A$  where the invariants in the proof are constructed from predicates in  $\mathcal{P}^*$ , the progress ranks are bounded, and the OR choices are uniform.*

In [2], the authors show completeness of predicate discovery for invariance properties (i.e.,  $\text{AG}(p)$ ) relative to an oracle which can widen intermediate results of the fixpoint computation of  $\Phi = \text{EF}(\neg p)$ , by dropping some conjunctive terms. Notice that the widened fixpoint,  $\Phi^+$ , is defined in terms of predicates from  $\mathcal{P}^*$ , and  $\neg(\Phi^+)$  is an inductive invariant implying  $\text{AG}(p)$ . Thus, a generalization of their result including existential and progress properties can be derived from these observations and Theorem 5. This method of proof also shows that the result holds for more powerful “clairvoyant” oracles, which may perform widening using predicates in  $\mathcal{P}^*$  that do not appear at the current stage.

**Theorem 6 (*Bisimulation and Finite-state Completeness*)** *If LTS  $M$  has a finite bisimulation quotient preserving the predicates,  $AP$ , in a property  $A$ , then predicate discovery coupled with abstraction produces a feasible result.*

**Proof Sketch.** The proof hinges on the fact that symbolic algorithms for bisimulation minimization (e.g., [5,25]) compute a finite quotient that is based on  $\mathcal{P}^*$  for a suitable initial choice of  $\mathcal{P}_0$ . Bisimulation ensures that the conditions in Theorem 5 on progress ranks and OR choice are met. Consequently, predicate discovery always produces a feasible abstraction for finite-state systems.  $\square$

## 5 Related Work

There is a large literature on the links between program abstraction and model checking – we discuss here only the most closely related results. As mentioned earlier, [32,22,23] show that simulation augmented with progress hints is complete for linear time properties, based on earlier work on deductive approaches [6,30]. Our abstraction method builds on this work, but uses progress hints differently, and handles branching time logics (such as the  $\mu$ -calculus), which are more powerful than LTL and include both universal and existential modalities. We also offer a finer analysis of completeness, with and without progress hints.

Another closely related line of research is the work in [10,13,14] on abstraction for the  $\mu$ -calculus and sub-logics, based on the abstract interpretation paradigm [12]. The abstract program has *two* transition relations, one used for checking existential properties, the other for universal ones. Our method uses similar  $\exists\exists$  and  $\forall\exists$  abstractions, but the duality is expressed locally by the alternation within the ATS. These papers study the precision of abstract interpretations, but not completeness. It is not known whether their abstraction methods are complete. From the results of this paper, however, this seems unlikely, since completeness seems to require both choice predicates and rank functions, applied in a manner closely tied to the property automaton. A completeness result is given in [9] for  $\text{CTL}^*$ , but only under a strict congruence assumption on the concrete system. Partial transition systems (e.g., those with may and must relations) have been used to define abstractions preserving branching time properties [7,18] – but with a “gap” where preservation is uncertain. Partial information is an orthogonal issue, and such methods can be incorporated in the analysis of ATS’s. We have related our completeness results to those in [2] in the previous section.

**Acknowledgements:** Thanks go to Dennis Dams for very helpful discussions in the course of this work, and to the referees for several insightful comments.

## References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *COMPOS*, volume 1536 of *LNCS*, 1997. Journal version in *JACM*.
2. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, number 2280 in *LNCS*, 2002.
3. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, 2001.
4. S. Bensalem, Y. Lakhnech, and S. Owre. InVeST: A tool for the verification of invariants. In *CAV*, volume 1427 of *LNCS*, 1998.
5. A. Bouajjani, J-C. Fernandez, and N. Halbwachs. Minimal model generation. In *CAV*, volume 531 of *LNCS*, 1990.
6. I.A. Browne, Z. Manna, and H.B. Sipma. Generalized temporal verification diagrams. In *FST&TCS*, volume 1026 of *LNCS*, 1995.
7. G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR*, volume 1877 of *LNCS*, 2000.

8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, 2000.
9. E.M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *TOPLAS*, 16(5), 1994.
10. R. Cleaveland, S. P. Iyer, and D. Yankelevich. Optimality in abstractions of model checking. In *SAS*, volume 983 of *LNCS*, 1995.
11. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, 2001.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1997.
13. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.
14. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *TOPLAS*, 19(2), 1997.
15. E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *CACM*, 18(8), 1975.
16. E. A. Emerson, C.S. Jutla, and A.P. Sistla. On model-checking for fragments of  $\mu$ -calculus. In *CAV*, 1993.
17. E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy (extended abstract). In *FOCS*, 1991.
18. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, volume 2154 of *LNCS*, 2001.
19. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
21. G. J. Holzmann and M. H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2), 2000.
22. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1), 2000.
23. Y. Kesten, A. Pnueli, and M. Vardi. Verification by augmented abstraction: The automata-theoretic view. *JCSS*, 62(4), 2001.
24. D. Kozen. Results on the propositional mu-calculus. In *ICALP*, 1982.
25. D. Lee and M. Yannakakis. Online minimization of transition systems. In *STOC*, 1992.
26. R. Milner. An algebraic definition of simulation between programs. In *2nd IJCAI*, 1971.
27. K. S. Namjoshi. Certifying model checkers. In *CAV*, number 2102 in *LNCS*, 2001.
28. K. S. Namjoshi. Lifting temporal proofs across abstractions. In *VMCAI*, volume 2575 of *LNCS*, 2003.
29. K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, volume 1855 of *LNCS*, 2000.
30. H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. *Formal Methods in System Design*, 15(1), 1999.
31. R.S. Streett and E.A. Emerson. The propositional mu-calculus is elementary. In *ICALP*, 1984. Full version in *Inf. and Comp.* 81(3), pp. 249-264, 1989.
32. T.E. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Stanford University, 1999.
33. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ICSE*, 2000.



# Certifying Optimality of State Estimation Programs

Grigore Roşu<sup>1</sup>, Ram Prasad Venkatesan<sup>1</sup>,  
Jon Whittle<sup>2</sup>, and Laurenţiu Leuştean<sup>3</sup>

<sup>1</sup> Department of Computer Science, University of Illinois at Urbana-Champaign

<sup>2</sup> NASA Ames Research Center, California

<sup>3</sup> National Institute for Research and Development in Informatics, Bucharest  
{grosu,rpvenkat}@cs.uiuc.edu, jonathw@email.arc.nasa.gov, leo@ici.ro

**Abstract.** The theme of this paper is certifying software for state estimation of dynamic systems, which is an important problem found in spacecraft, aircraft, geophysical, and in many other applications. The common way to solve state estimation problems is to use *Kalman filters*, i.e., stochastic, recursive algorithms providing statistically optimal state estimates based on noisy sensor measurements. We present an optimality certifier for Kalman filter programs, which is a system taking a program claiming to implement a given formally specified Kalman filter, as well as a formal certificate in the form of assertions and proof scripts merged within the program via annotations, and tells whether the code correctly implements the specified state estimation problem. Kalman filter specifications and certificates can be either produced manually by expert users or can be generated automatically: we also present our first steps in merging our certifying technology with AUTOFILTER, a NASA Ames state estimation program synthesis system, the idea being that AUTOFILTER synthesizes proof certificates together with the code.

## 1 Introduction

A common software development task in the spacecraft navigation domain is to design a system that can estimate the attitude of a spacecraft. This is typically mission-critical software because an accurate attitude estimate is necessary for the spacecraft controller to tilt the craft's solar panels towards the sun. Attitude estimators for different spacecraft, as well as a spectrum of other estimators for dynamic systems in general, are typically variations on a theme, solving a *state estimation problem*. The common way to solve state estimation problems is to use *Kalman filters*. A Kalman filter is essentially a set of mathematical equations implementing a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated error between the real and the predicted states. Since the time of their introduction [8] in 1960, Kalman filters have been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation. This is likely due in large part not only to

advances in digital computing that made their use practical, but also to the relative simplicity and robust nature of the filter itself.

Despite their apparent simplicity, implementations of state estimation problems are quite hard to prove correct. Correctness in this context means that the state calculated by a given implementation, called the state estimate, is mathematically optimal when one considers all the hypotheses given in the description of the state estimation problem. Descriptions of such problems are given as sets of stochastic difference equations, and optimality is rigorously, statistically formulated using matrix differentiation. Since this domain is not common in computer-aided verification, we find it appropriate to dedicate an entire section, Section 2, to Kalman filters, their subtleties (especially the assumptions under which they can be proved optimal), as well as to their variations.

Due to the need of accurate state estimates in critical applications, it is important to be able to provide optimality guarantees whenever possible. In this paper we present work in progress on a verification environment for the nontrivial class of domain-specific programs solving state estimation problems. Our long term goal is to provide a domain formalization, including axiomatization of several segments of mathematics together with significant lemmas, and a friendly tool making use of it to formally certify optimality. Up to this moment we were able to develop a common framework and a prototype tool with which we were able to certify three of the most important Kalman filters, the simple Kalman filter, the information filter and the extended Kalman filter. One would, of course, like to certify software as automatically as possible, but this is very rarely feasible due to intractability arguments and clearly close to impossible for the complex domain presented in this paper. Therefore, user intervention is often needed to insert domain-specific knowledge into the programs to be certified, usually under the form of code annotations. The certifier in this paper needs annotations for model specifications, assertions, and proof scripts. It is mostly implemented in Maude [5], a freely distributed high-performance executable specification system in the OBJ [7] family, supporting both rewriting logic [11] and membership equational logic [12]. Because of its efficient rewriting engine and because of its metalanguage and modularization features, Maude is an excellent tool to develop executable environments for various logics, models of computation, theorem provers, and even programming languages. The work in this paper falls under what was called *domain-specific certification* in [10].

A growth area in the last couple of decades has been code generation. Although commercial code generators are mostly limited to generating stub codes from high level models (e.g., in UML), program synthesis systems that can generate fully executable code from high level behavioral specifications are rapidly maturing (see, for example, [20,18]), in some cases to the point of commercialization (e.g., SciNapse [1]). In program synthesis, there is potential for automatically verifying nontrivial properties because additional background information – from the specification and the synthesis knowledge base – is available. Following the ideas in [16,15], we show how we coupled together AUTOFILTER, a NASA Ames synthesis system for the state estimation domain, and our prototype certifier.

The main idea here is to modify AUTOFILTER to synthesize not only the code, but also the appropriate formal annotations needed by the certifier.

Due to space limitation, we only give a high level overview of our work in optimality certification of state estimates. The reader is referred to a 50 page report [9] presenting in detail a previous version of this work. Important related work includes proof-carrying code [14] and extended static checking [3,17].

## 2 Kalman Filters

A Kalman filter is essentially a set of mathematical equations implementing a predictor-corrector type estimator that is *optimal* in the sense that it minimizes the estimated *error* covariance - when some assumptions are met. Since the time of their introduction [8], Kalman filters have been the subject of extensive research and application, particularly in the area of autonomous or assisted navigation. This is likely due in large part to advances in digital computing that made their use practical, but also to the relative simplicity and robust nature of the filter itself. We have tested our state estimation certification technique on three Kalman filters in current use, the simple Kalman filter, the information filter and the extended Kalman filter, which are briefly discussed next.

The *simple Kalman filter* addresses the general problem of estimating the state  $x \in \mathbb{R}^n$  of a discrete-time controlled system that is governed by the linear stochastic difference equation for  $x$  with measurement  $z \in \mathbb{R}^m$ :

$$x_{k+1} = \Phi_k x_k + w_k \quad (1) \quad z_k = H_k x_k + v_k. \quad (2)$$

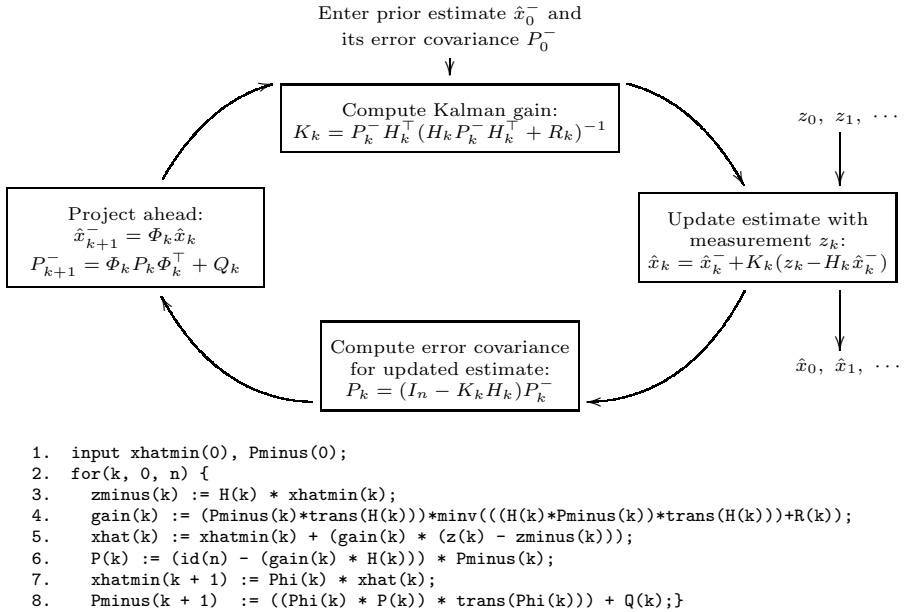
$x_k$  is the process state vector at time  $k$ . For example, the state vector  $x_k$  might contain three variables representing the rotation angles of a spacecraft. Equation (1) is the *process model*, describing the state dynamics over time - the state at time  $k + 1$  is obtained by multiplying the state transition matrix  $\Phi_k$  by the previous state  $x_k$ . The model is imperfect, however, as represented by the addition of the process noise vector  $w_k$ . Equation (2) is the *measurement model* and models the relationship between the measurements and the state. This is necessary because the state usually cannot be measured directly. The measurement vector,  $z_k$ , is related to the state by matrix  $H_k$ . The random vectors  $w_k$  and  $v_k$  represent the process and measurement noise, respectively, and they are assumed to be independent of each other, white, and with normal distribution:

$$\begin{aligned} p(w_k) &\sim N(0, Q_k) & (3) & \quad E[w_k w_i^\top] = \begin{cases} Q_k, & \text{if } i = k \\ 0, & \text{if } i \neq k \end{cases} & (6) \\ p(v_k) &\sim N(0, R_k) & (4) & \\ E[w_k v_i^\top] &= 0 & (5) & \quad E[v_k v_i^\top] = \begin{cases} R_k, & \text{if } i = k \\ 0, & \text{if } i \neq k. \end{cases} & (7) \end{aligned}$$

As an example of how the simple Kalman filter works in practice, consider a simple spacecraft attitude estimation problem. Attitude is usually measured using gyroscopes, but the performance of gyroscopes degrades over time so the error in the gyroscopes is corrected using other measurements, e.g., from a star tracker. In this formulation, the process equation (1) would model how the gyroscopes

degrade and the equation (2) would model the relationship between the star tracker measurements and the three rotation angles that form the state (in this case,  $H_k$  would be the identity matrix because star trackers measure rotation angles directly). From these models, a Kalman filter implementation would produce an optimal estimate of the current attitude, where the uncertainties in the problem (gyro degradation, star tracker noise, etc.) have been minimized.

Before we present the implementation of the simple Kalman filter, we need several important notions. Let us define  $\hat{x}_k^- \in \mathbb{R}^n$  to be the *a priori state estimate* at step  $k$  given knowledge of the process prior to step  $k$ , and  $\hat{x}_k \in \mathbb{R}^n$  be the *a posteriori state estimate* at step  $k$  given measurement  $z_k$ , with their *a priori* and *a posteriori estimate errors*  $e_k^- = x_k - \hat{x}_k^-$  and  $e_k = x_k - \hat{x}_k$ , respectively. The *a priori estimate error covariance* is then  $P_k^- = E[e_k^-(e_k^-)^\top] = E[(x_k - \hat{x}_k^-)(x_k - \hat{x}_k^-)^\top]$  and the *a posteriori estimate error covariance* is  $P_k = E[e_k e_k^\top] = E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^\top]$ . We define also the *measurement prediction* as  $z_k^- = H_k \hat{x}_k^-$ .



**Fig. 1.** Simple Kalman filter loop and intermediate code.

In deriving the equations for the Kalman filter program, one needs to first find an equation that computes an a posteriori estimate  $\hat{x}_k$  as a linear combination of an a priori estimate  $\hat{x}_k^-$  and a weighted difference between the actual measurement  $z_k$  and the measurement prediction  $z_k^-$  as shown below:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - z_k^-). \quad (8)$$

where  $(z_k - z_k^-)$  is called the *measurement innovation*, or the *residual*, and reflects the discrepancy between the predicted and the actual measurements. The  $n \times m$

matrix  $K_k$  is chosen to be the *gain*, or *blending factor*, that minimizes the a posteriori estimate error covariance  $P_k$ . One form of the Kalman gain is

$$K_k = P_k^- H_k^\top (H_k P_k^- H_k^\top + R_k)^{-1}, \quad (9)$$

which, after hundreds of basic equational steps, yields the covariance matrix  $P_k = (I_n - K_k H_k) P_k^-$ . Figure 1 gives a complete picture of the simple Kalman filter, both as a diagram and as intermediate code that AUTOFILTER generates.

AUTOFILTER takes as input a mathematical specification including equations (1) - (7) and also descriptions of the noise characteristics and filter parameters, and first generates code like in Figure 1 and then translates it into C++, Matlab or Octave. In this paper we only consider code in the intermediate language. Despite its apparent simplicity, the proof of optimality for the simple Kalman filter is quite complex. The main proof task is to show that the vector  $\hat{x}_k$  is the best estimate of the state vector  $x_k$  at time  $k$ , under appropriate simplifying assumptions, and is usually presented in books informally on several pages (see [2], for example). In the sequel, we sketch this proof, emphasizing those aspects which are particularly relevant for its mechanization, especially the *assumptions*.

The very first assumption is that  $\hat{x}_0^-$  and  $P_0^-$  are the best initial prior estimate and its error covariance matrix. Another assumption is that the measurement prediction at any given time  $k$ ,  $z_k^-$ , is the most probable measurement. The most important assumption says that the best estimate  $\hat{x}_k$  is a *linear* blending of the residual and the prior estimate (8). The justification for this assumption is rooted in the probability of the prior estimate  $\hat{x}_k^-$  conditioned on all the prior measurements  $z_k$  (see [2,19] for more details). Formally, this says that the best estimate  $\hat{x}_k$  is somewhere in the image of the function  $\hat{x}_k(y) := \lambda y \cdot (\hat{x}_k^- + y(z_k - z_k^-))$ , where the blending factor  $y$  is an  $n \times m$  matrix. If  $P_k(y) := E[(x_k - \hat{x}_k(y))(x_k - \hat{x}_k(y))^\top]$  is the a posteriori error covariance matrix regarded as a function of  $y$ , then we wish to find the particular  $y$  that minimizes the individual terms along the major diagonal of  $P_k(y)$ , because these terms represent the estimation error covariances for the elements of the state vector being estimated. Using another assumption, that the individual mean-square errors are also minimized when the total is minimized, our problem reduces to finding the  $y$  that minimizes the trace,  $\text{trace}(P_k(y))$ , of  $P_k(y)$ , where the trace of a square matrix is the sum of the elements on its major diagonal. This optimization is done using a differential calculus approach. Differentiation of matrix functions is a complex field that we partially formalized and which we cannot cover here, but it is worth mentioning, in order for the reader to anticipate the non-triviality of this proof, that the  $y$  we are looking for is the solution of the equation  $d(\text{trace}(P_k(y)))/dy = 0$ , where for a standard function  $f(y_{11}, y_{12}, \dots)$  on the elements of the matrix  $y$ , such as  $\text{trace}(P_k(y))$ , its derivative  $df/dy$  is the matrix  $(df/dy_{ij})_{ij}$  having the same dimension  $n \times m$  as  $y$ . Using two important differentiation lemmas, namely “ $d(\text{trace}(yA))/dy = A^\top$  if  $yA$  is a square matrix” and “ $d(\text{trace}(yAy^\top))/dy = 2yA$  if  $A$  is a symmetric matrix”, after several thousands of basic proof steps one gets the desired solution, the so called Kalman gain (9). The a posteriori best estimate,  $\hat{x}_k := \hat{x}_k(K_k)$ , and the covariance ma-

trix associated with the optimal estimate,  $P_k(K_k)$ , can now also be calculated by equational reasoning, and the updated estimate  $\hat{x}_k$  is “projected ahead” via the transition matrix,  $\hat{x}_{k+1}^- = \Phi_k \hat{x}_k$ . The fact  $\hat{x}_{k+1}^-$  is the best prior estimate at time  $k + 1$  follows by another important assumption, saying that the best prior estimate at the next step follows the state equation (1) using the best estimate at the current state, but where the contribution of the process noise  $w_k$  is ignored (we are justified in doing this because the noise  $w_k$  has zero mean and is not correlated with any one of the previous  $w$ ’s). By equational reasoning it follows now that  $P_{k+1}^-$ , the error covariance matrix of  $\hat{x}_{k+1}^-$ , is  $\Phi_k P_k \Phi_k^\top + Q_k$ .

The flow of calculations above is for simple Kalman filters, but its equations can be algebraically manipulated into a variety of forms. An alternative form is known as the *information filter* [2], which additionally assumes that the matrices  $P_k^-$ ,  $P_k$ , and  $R_k$  admit inverses.  $(P_k^-)^{-1}$  is thought of as a measure of the information content of the a priori estimate. Then  $P_k = \infty$ , i.e.,  $(P_k^-)^{-1} = 0$ , corresponds to infinite uncertainty, or zero information. This leads to the indeterminate form  $\frac{\infty}{\infty}$  in the Kalman gain expression (9), so one can not apply the simple Kalman filter due to rounding errors. The information filter accommodates this situation and is based on the equations (which can be obtained as above):

$$P_k^{-1} = (P_k^-)^{-1} + H_k^\top R_k^{-1} H_k \quad (10) \quad K_k = P_k H_k^\top R_k^{-1}. \quad (11)$$

The Kalman filters discussed so far address the general problem of estimating the state  $x \in \mathbb{R}^n$  of a discrete-time controlled process that is governed by a *linear* stochastic difference equation. However, some of the most interesting and successful applications of Kalman filters are *non-linear*, i.e., the process and measurement models are given by equations of the form

$$x_{k+1} = f(x_k, u_k) + w_k \quad (12) \quad z_k = h(x_k) + v_k, \quad (13)$$

where  $f$  and  $h$  are non-linear functions,  $u_k$  is a deterministic forcing function (regard it as an input), and the random vectors  $w_k$  and  $v_k$  again represent the process and the measurement noise and satisfy the same conditions as for the simple Kalman filter. To simplify computations and to make the problem implementable, one can linearize it about a trajectory that is continually updated with the state estimates resulting from the measurements. The new filter obtained this way is called *extended Kalman filter* (or simply *EKF*). Since the noises  $w_k$  and  $v_k$  have mean 0 and since  $\hat{x}_k$  and  $\hat{x}_k^-$  are estimates anyway, one can approximate the a priori state estimate and measurement prediction vectors as  $\hat{x}_{k+1}^- = f(\hat{x}_k, u_k)$ , and  $z_k^- = h(\hat{x}_k^-)$ , respectively. Taking  $\Phi_k$  and  $H_k$  the Jacobian matrices

$$\Phi_k = \left( \frac{\delta f_i}{\delta x_j}(\hat{x}_k, u_k) \right)_{i,j}, \quad H_k = \left( \frac{\delta h_i}{\delta x_j}(\hat{x}_k^-) \right)_{i,j},$$

one can approximate  $f$  and  $h$  with their first order Taylor series expansions

$$f(x_k, u_k) = f(\hat{x}_k, u_k) + \Phi_k(x_k - \hat{x}_k), \quad h(x_k) = h(\hat{x}_k^-) + H_k(x_k - \hat{x}_k^-).$$

One can get the new governing equations that linearize an estimate

$$x_{k+1} = \hat{x}_{k+1}^- + \Phi_k(x_k - \hat{x}_k^-) + w_k, \quad z_k = z_k^- + H_k(x_k - \hat{x}_k^-) + v_k,$$

which can be solved and implemented following closely the simple Kalman filter.

### 3 Specifying and Annotating Kalman Filters

In order to perform computer-aided optimality certification of programs implementing sophisticated state estimation problems like the ones above, one first needs to rigorously specify the statistical problem together with all its needed *assumptions*. In fact, an initially unexpected important benefit of our technique, whose value we discovered progressively during experiments, is that it makes our user aware of all the assumptions under which a Kalman filter program indeed calculates the expected optimum state estimate, which usually include strictly those mentioned by authors of theorems in textbooks<sup>1</sup>. These specifications are defined on top of an axiomatically formalized abstract domain knowledge theory, including matrices, differentiation and probability theory (presented in the next section), and use Maude membership equational logic notation [6,5], which is natural but we do not explain here. The simple Kalman filter, for example, has 45 axioms (properties and assumptions); we comment on a few of these next.

Operations or constants declaring the matrices and vectors involved together with their dimensions are defined first:

```
ops x z v w : Nat -> RandomMatrix .
ops n m p : -> Nat .
eq row(x(K)) = n .   eq column(x(K)) = 1 .
eq row(Phi(K)) = n . eq column(Phi(K)) = n .
eq row(v(K)) = m .   eq column(v(K)) = 1 .
...

ops R Q Phi H : Nat -> Matrix .
var K : Nat .
eq row(w(K)) = n .   eq column(w(K)) = 1 .
eq row(z(K)) = m .   eq column(z(K)) = 1 .
eq row(H(K)) = m .   eq column(H(K)) = n .
```

The sort `Nat` comes from a Maude builtin module, while the sorts `Matrix` and `RandomMatrix` are defined within the abstract domain, presented in the next section, and stay for random matrix variables and for matrices, respectively. Other axioms specify model equations (which are labeled for further use), such as

```
[ax*> | KF-next]      eq x(K + 1) = Phi(K) * x(K) + w(K) .
[ax*> | KF-measurement] eq z(K) = H(K) * x(K) + v(K) .
[ax*> | RK]           eq R(K) = E| v(K) * trans(v(K)) | .
[ax*> | QK]           eq Q(K) = E| w(K) * trans(w(K)) | .
```

Operations `_*`, `_+_`, `trans` (matrix transpose) and `E|_` (error covariance) on matrices or random matrices are all axiomatized in the abstract domain. Other assumptions that we do not formalize here due to space limitation include independence of noise, and the fact that the best prior estimate at time  $k + 1$  is the product between  $\Phi(k)$  and the best estimate calculated previously at step  $k$ . One major problem that we encountered while developing our proofs following textbook proofs was that these assumptions, and many others not mentioned here, are so well and easily accepted by experts that they don't even make their

<sup>1</sup> Which is understandable, otherwise theorems would look too heavy for humans.

use explicit in proofs. One cannot do this in formal proving, so one has the nontrivial task to detect and then declare them explicitly as special axioms.

In order to machine check proofs of optimality, they must be properly decomposed and linked to the actual code. This can be done in many different ways. For simplicity, we prefer to keep everything in one file. This can be done by adding the specification of the statistical model at the beginning of the code, and then by adding appropriate formal statements, or assertions, as annotations between instructions, so that one can prove the next assertion from the previous ones and the previous code. Formal proofs are also added as annotations where needed. Notice that by “proof” we here mean a series of *hints* that allows our proof assistant and theorem prover, Maude’s Inductive Theorem Prover (ITP) [4], to generate and then check the detailed proof. The next shows how the 8 line simple Kalman filter code in Figure 1 is annotated in order to be automatically verifiable by our optimality certifier. In order to keep the notation simple, we either removed or replaced by plain English descriptions the formal specification, assertions and proofs occurring in the code as comments. It may be worth mentioning that the entire annotated simple Kalman filter code has about 300 lines, that it compresses more than 100,000 basic proof steps as generated by ITP, and that it takes about 1 minute on a 2.4GHz standard PC to generate all the proofs from their ITP hints and then check them:

```

/* Specification of the state estimation problem ... about 45 axioms/assumptions */
1.  input xhatmin(0), Pminus(0);
/* Proof assertion 1: ... */
/* Assertion 1: xhatmin(0) and Pminus(0) are best prior estimate and its error covar. */
2.  for(k,0,n) {
/* Assertion 2: xhatmin(k) and Pminus(k) are best prior estimate and its error covar. */
3.    zminus(k) := H(k) * xhatmin(k);
4.    gain(k) := Pminus(k) * trans(H(k)) * minv(H(k) * Pminus(k) * trans(H(k)) + R(k));
/* Proof assertion 3: ... */
/* Assertion 3: gain(k) minimizes the error covariance matrix */
5.    xhat(k) := xhatmin(k) + (gain(k) * (z(k) - zminus(k)));
/* Proof assertion 4: ... */
/* Assertion 4: (the main goal) xhat(k) is the best estimate */
6.    P(k) := (id(n) - (gain(k) * H(k))) * Pminus(k);
/* Proof assertion 5: ... */
/* Assertion 5: P(k) is the error covariance matrix of xhat(k) */
7.    xhatmin(k + 1) := Phi(k) * xhat(k);
8.    P(k + 1) := ((Phi(k) * P(k)) * trans(Phi(k))) + Q(k);
/* Proof assertion 2 at time k + 1: ... */
}

```

The proof assertions and proofs above should be read as follows: proof assertion  $n$  is a proof of assertion  $n$  in its current environment. The best we can assert between instructions 1 and 2 is that `xhatmin(0)` and `Pminus(0)` are initially the best prior estimate and error covariance matrix, respectively. This assertion is an assumption in the theory of Kalman filters, axiom in our specification, so it can be immediately checked. Between 2 and 3 we assert that `xhatmin(k)` and `Pminus(k)` are the best prior estimate and its error covariance matrix, respectively. This is obvious for the first iteration of the loop, but needs to be proved for the other iterations. Therefore, our certifier performs an implicit proof by induction. The assertion after line labeled 4 is that `gain(k)` minimizes the a posteriori error covariance matrix. This was the part of the proof that was the most difficult to formalize. We show it below together with its corresponding ITP proof script:



```

[proof assertion-3]
(set (instruction-1-1 instruction-2 assertion-1-1 assertion-1-2 Estimate KF-measurement
    zero-cross_v_x-xhatmin_1 zero-cross_v_x-xhatmin_2 RK id*left id*right id-trans
    distr*trans minus-def1 misc-is E- E+ E*left E*right trans-E fun-scalar-mult
    fun-mult fun-trans fun-def1 fun-def2 minimizes trace+ trace- tracem-def1
    tracem-def2 trace-lemma1 trace-lemma1* trace-lemma2 lemma-1 lemma-2 lemma-3
    lemma-4 lemma-5 lemma-6 lemma-7 lemma-8 lemma-9 lemma-10 lemma-11) in (1) .)
(rwr (1) .)
(idt (1) .)
----- */
/* -----
[assertion-3]
    eq gain(K) minimizes /\ y . (E|(x(K) - Estimate(K,y)) * trans(x(K) - Estimate(K,y)))|)
    = (true) .
----- */
    
```

Hence, we first “set” 44 axioms, lemmas, and/or previous assertions or instructions, all referred to by their labels, and then simplify the proof task by rewriting with the ITP command `rwr`. This particular proof can be done automatically (`idt` just checks for identity). The axioms are either part of the particular Kalman filter specification under consideration (such as those on the second row) or part of the axiomatization of the abstract domain (such as those on the third and fourth rows), which is general to all Kalman filters. The lemmas are essentially properties of the state estimation domain, so they belong to the abstract domain. As explained in the next section, these lemmas have been devised by analyzing several concrete state estimation problems and extracting their common features.

## 4 Specifying the Abstract Domain Knowledge

To generate and automatically certify the optimality proofs discussed so far, one needs to first formalize the state estimation *domain knowledge*, which includes matrices, random matrices, functions on matrices, and matrix differentiation. Formalizing the abstract domain was by far the most tedious part of this project, because it suffered a long series of refinements and changes as new and more sophisticated state estimation problems were considered. Since most of the operations on matrices are partial, since domain specifications are supposed to be validated by experts and since our work is highly experimental at this stage, we decided to use Maude [5] and its ITP tool [4] to specify and prove properties of our current domain knowledge, because these systems provide implicit strong support for partiality<sup>2</sup> (via memberships), their specifications are human readable due to the mix-fix notation, and can be easily adapted or modified to fulfill our continuously changing technical needs. Our current domain passed the criterion telling if a total theory can be safely regarded as partial [13]. However, we think that essentially any specification language could be used instead, if sufficient precautions are taken to deal properly with partial operations.

**Matrices and Random Matrices.** Matrices are extensively used in all state estimation problems and their optimality proofs, so we present their formalization first. Four sorts, in the subsort lattice relationship (using Maude notation)

<sup>2</sup> We are not aware of any systems providing explicit support for partiality.

“`subsorts Matrix < MatrixExp RandomMatrix < RandomMatrixExp`”, have been introduced. Random matrices are matrices whose elements can be random variables, such as the state to be estimated, the measurements and/or the noise, and (random) matrix expressions can be formed using matrix operators. Most of the operations and axioms/lemmas in matrix theory are *partial*. For example, multiplication is defined iff the number of columns of the first matrix equals the number of rows of the second. It is a big benefit, if not the biggest, that Maude provides support for partiality, thus allowing us to compactly specify matrix theory and do partial proofs. The resulting sort (or rather “kind”) of a partial operator is declared between brackets in Maude; for example, the partial operation of multiplication is defined as “`op _*_ : MatrixExp MatrixExp -> [MatrixExp]`”. Transpose of a matrix is total, so it is defined as “`op trans : MatrixExp -> MatrixExp`”. Membership assertions, stating when terms have “proper” sorts, can now be used to say when a partial operation on matrices is defined (two total operators, “`ops row column : MatrixExp -> Nat`”, are needed). For example, the following axioms define multiplication together with its dimensions:

```
vars P Q R : MatrixExp .                cmb P*Q : MatrixExp if column(P) == row(Q).
ceq column(P*Q) = column(Q) if P*Q : MatrixExp .  ceq row(P*Q) = row(P) if P*Q : MatrixExp .
```

Most of the matrix operators are *overloaded*, i.e., defined on both matrices and random matrices. Operators relating the two, such as the error covariance operator “`op E|_ : RandomMatrixExp -> MatrixExp`”, together with plenty of axioms relating the various operators on matrices, such as distributivity, transpose of multiplications, etc., are part of the abstract domain theory.

**Functions on Matrices.** An important step in state estimation optimality proofs (see Section 2) is that the best estimate is a linear combination of the best prior estimate and the residual (8). The coefficient of this linear dependency is calculated such that the error covariance  $P_k$  is minimized. Therefore, before the optimal coefficient is calculated, and in order to calculate it, the best estimate vector is regarded as a *function* of the form  $\lambda y.(\langle \text{prior} \rangle + y * \langle \text{residual} \rangle)$ . In order for this function to be well defined,  $y$  must be a matrix having proper dimensions. We formally define functions on matrices and their properties by declaring new sorts, `MatrixVar` and `MatrixFun`, together with operations for defining functions and for applying them, respectively:

```
op /\_.. : MatrixVar MatrixExp -> MatrixFun .
op _.. : MatrixFun MatrixExp -> [MatrixExp] .
cmb (/\ X . P)(R) : MatrixExp if X + R : MatrixExp .
```

Several axioms on functions are defined, such as

```
ceq (/\X.X)(R) = R if X + R : MatrixExp .
ceq (/\X.(P+Q))(R) = (/\X.P)(R) + (/\X.Q)(R) if X + R : MatrixExp and P + Q : MatrixExp .
```

**Matrix Differentiation.** As shown in Section 2, in order to prove that  $\hat{x}_k$  is the best estimate of the state vector  $x_k$ , a differential calculus approach is used. Axiomatization matrix differentiation can be arbitrarily complicated; our approach is top-down, i.e., we first define properties *by need*, use them,

and then prove them from more basic properties as appropriate. For example, the only property used so far linking optimality to differentiation is that  $K$  minimizes  $\lambda y.P$  iff  $(d(\text{trace}(\lambda y.P))/dy)(K) = 0$ . For that reason, to avoid deep axiomatizability of mathematics, we only defined a “derivative” operation “`op d|trace_|/d_ : MatrixFun MatrixVar -> MatrixFun`” with axioms like:

```
ceq (d|trace(/\X.X)|/d(X))(R) = id(row(X)) if X + R : MatrixExp .
ceq (d|trace(/\X.(P+Q))|/d(X))(R) = (d|trace(/\X.P)|/d(X))(R) + (d|trace(/\X.Q)|/d(X))(R)
  if X + R : MatrixExp and P + Q : MatrixExp .
```

The two important lemmas used in Section 2 are also added as axioms:

```
ceq d|trace(/\X.(X*P))|/d(X) = /\X.trans(P) if X * P : MatrixExp and not(X in P) .
ceq d|trace(/\X.(X*P*trans(X))|/d(X) = /\X.(2*X*P)
  if X * P : MatrixExp and P * trans(X) : MatrixExp and trans(P) == P and not(X in P) .
```

**Domain-Specific Lemmas.** One could, of course, prove the properties above from more basic properties of reals, traces, functions and differentiations, but one would need to add a significant body of mathematical knowledge to the system. It actually became clear at an early stage in the project that a database of domain-specific lemmas was needed. On the one hand, lemmas allow one to device more compact, modular and efficiently checkable proofs. On the other hand, by using a large amount of lemmas, the amount of knowledge on which our certification system is based can be reduced to just a few axioms of real numbers, so the entire system can be more easily validated and therefore accepted by domain experts. We currently have only 34 domain-specific lemmas, but their number is growing fast, as we certify more state estimation problems and refine the axiomatization of the abstract domain. These lemmas were proved using ITP [4], and were stored together with their proofs in a special directory. The user refers to a lemma by its unique label, just as to any axiom in the domain, and the certifier uses their proofs to synthesize and check the optimality proof.

## 5 Certifying Annotated Kalman Filters

There are several types and levels of certification, including testing and human code review. In this paper we address certification of programs for conformance with domain-specific properties. We certify that the computation flow of a state estimation program leads to an optimum solution. The reader should not get trapped by thinking that the program is thus “100% correct”. There can be round-off or overflow errors, or even violations of basic safety policies, e.g., when a matrix uses the metric measurement unit system and another uses the English system, which our certifier cannot catch. What we guarantee is that, under the given hypotheses (i.e., the specification at the beginning of the code), the mathematics underlying the given code provably calculates the best state estimate of the specified dynamic system. The certifier presented next uses a combination of theorem proving and proof checking; the interested reader is encouraged to download the certifier and its manual from <http://fsl.cs.uiuc.edu>.

**Cleaning the Code.** The certifier first removes the insignificant details from the code, including empty blocks, comments that bear no relevance to the domain and statements that initialize the variables. We are justified in doing this because we are interested in the correctness of the mathematical flow of computation and not in the concrete values of the variables or matrices.

**Generating Proof Tasks.** The cleaned annotated Kalman filter, besides code and specification, contains assertions and proof scripts. By analyzing their labels and positions, the certifier generates a set of proof tasks. This is a technical process whose details will appear elsewhere, but the idea is that a task “ $Domain + Spec + Before_A \models A$ ” is generated for each assertion  $A$ , where  $Before_A$  is the knowledge accumulated before  $A$  is reached in the execution flow. Each generated task is placed in a separate file, together with an *expanded proof*, in ITP [4] notation. The expanded proofs are generated from the original proof scripts (which refer to lemmas via their names), by replacing each use of a lemma by an instance of its proof<sup>3</sup>, which is taken from the database of domain-specific lemmas. The reason for doing so is based on the belief that certifying authorities have no reason to trust complex systems like ITP, but rather use their own simple proof checkers; in this case we would use ITP as a *proof synthesizer*.

**Proof Checking.** At this moment, however, we use ITP both as a proof generator and as a proof checker. More precisely, the certifier sends each proof task to ITP for validation. ITP executes the commands in the provided (expanded) proof script, and logs its execution trace in another file, so a skeptical user can double check it, potentially using a different checker.

**The Tool.** The certifier is invoked with the command `certifier [-cgv] pgm`. By default, it cleans, generates proof tasks and then verifies the annotated Kalman filter input. The cleaned code and the generated proof tasks by default are saved in appropriate files for potential further interest. Each of the options disable a corresponding action: `-c` disables saving the cleaned version, `-g` the generated proof tasks, and `-v` the verification step. Thus, `certifier -cv rover.code` would only generate the proof tasks associated to the state estimation code `rover.code`.

## 6 Synthesizing Annotated Kalman Filters

Like in proof-carrying code [14], the burden of producing the assertions and their proof scripts falls entirely on code producer’s shoulders. Despite the relative support provided by proof-assistants like ITP, this can still be quite inconvenient if the producer of the code is a human. Not the same can be said if the producer of the code is another computer program. In this section we present our efforts in merging the discussed certification technology with a NASA Ames state estimation program synthesis system, called AUTOFILTER, thus underlying the foundations of what we called *certifiable program synthesis* in [16,15].

<sup>3</sup> This is similar in spirit to what is also known as “cut elimination”.

AUTOFILTER takes a detailed specification of a state estimation problem as input, and generates intermediate code like the one in Figure 1 which is further transformed into C++, Matlab or Octave. It is built on an *algorithm schema* idea. A schema is a generic representation of a well-known algorithm. Most generally, it is a high-level description of a program which captures the essential algorithmic steps but does not necessarily carry out the computations for each step. In AUTOFILTER, a schema includes assumptions, applicability conditions, a *template* that describes the key steps of the algorithm, and the *body* of the schema which instantiates the template. Assumptions are inherent limitations of the algorithm and appear as comments in the generated code. Applicability conditions can be used to choose between alternative schemas. Note, however, that different schemas can apply to the same problem, possibly in different ways. This leads to choice points which are explored in a depth-first manner. Whenever a dead-end is encountered (i.e., an incomplete code fragment has been generated but no schema is applicable), AUTOFILTER backtracks, thus allowing it to generate multiple program variants for the same problem.

The main idea underlying the concept of certifiable program synthesis is to make a synthesis system generate not only code, but also *checkable correctness certificates*. In principle, this should be possible because any synthesis system worth its salt generates code from a logical specification of a problem, by performing formal reasoning, so it must be able to answer the question “why?” rigorously when it generates a certain block of code; otherwise, there is no legitimate reason to trust such a system. We are currently modifying AUTOFILTER to generate annotations together with its state estimation programs, in a form which is certifiable by the tool presented in the previous section. We are currently able to automatically certify any synthesized program which is an instance of a simple Kalman filter, and provide a general mechanism to extend it to all algorithm schemas. The annotations are stored with the program schemas at the *template* or *body* level. The template contains annotations that are global to the algorithm represented by that schema, e.g., the specification of the filter. The body contains annotations local to a particular piece of intermediate code used to instantiate the template, e.g., an assertion that `gain(k)` minimizes the covariance matrix *for a particular instantiation of the gain matrix and the covariance matrix*. Since AUTOFILTER can generate multiple implementations of each schema, by attaching annotations to a schema, it can generate annotations for each variation. The annotations of each schema are added by experts, who essentially formally prove each schema correct with respect to its hypotheses. The advantage of synthesis in this framework is that these hard proofs are done *only once* and then instantiated by the synthesis engine whenever needed.

## References

1. R. Akers, E. Kant, C.Randall, S. Steinberg, and R.Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Computational Science and Engineering*, 4(3):32–42, 1997.

2. R.G. Brown and P. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Son, 3rd edition, 1997.
3. Compaq : Extended Static Checking. <http://www.research.compaq.com/SRC/esc>.
4. M Clavel. ITP tool. Department of Philosophy, University of Navarre, <http://sophia.unav.es/~clavel/itp/>.
5. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In *Proceedings of WRLA'06*, volume 4 of *ENTCS*. Elsevier, 1996.
7. J.Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer, 2000.
8. R.E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME-Journal of Basic Engineering*, 82:35–45, 1960.
9. Laurențiu Leuştean and Grigore Roşu. Certifying Kalman Filters. Technical Report TR 03-02, RIACS, 2003.
10. M. Lowry, T. Pressburger, and G.Roşu. Certifying domain-specific policies. In *Proceedings of ASE'01*, pages 81–90. IEEE, 2001. Coronado Island, California.
11. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, pages 73–155, 1992.
12. J. Meseguer. Membership algebra as a logical framework for equational specification. In *Proceedings of WADT'97*, volume 1376 of *LNCS*, pages 18–61, 1998.
13. J. Meseguer and G. Roşu. A total approach to partial algebraic specification. In *Proceedings of ICALP'02*, volume 2380 of *LNCS*, pages 572–584, 2002.
14. G.C. Necula. Proof-carrying code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.
15. G. Roşu and J. Whittle. Towards certifying domain-specific properties of synthesized code. In *Proceedings, Verification and Computational Logic (VCL'02)*, 2002. Pittsburgh, PA, 5 October 2002.
16. G. Roşu and J. Whittle. Towards certifying domain-specific properties of synthesized code (extended abstract). In *Proceedings of ASE'02*. IEEE, 2002.
17. K. Rustan, M. Leino, and Greg Nelson. An extended static checker for modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *LNCS*, pages 302–305. Springer, April 1998.
18. Y. V. Srinivas and R. Jüllig. Specware: Formal support for composing software. In Bernhard Moller, editor, *Mathematics of Program Construction: third international conference, MPC '95*, volume 947 of *LNCS*, Kloster Irsee, Germany, 1995. Springer.
19. G. Welch and G. Bishop. An Introduction to the Kalman Filter. Course, SIGGRAPH 2001.
20. J. Whittle, J. van Baalen, J. Schumann, P. Robinson, T. Pressburger, J. Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proceedings of ASE'01*, San Diego, CA, USA, 2001.

# Domain-Specific Optimization in Automata Learning

Hardi Hungar, Oliver Niese, and Bernhard Steffen

University of Dortmund

{Hardi.Hungar, Oliver.Niese, Bernhard.Steffen}@udo.edu

**Abstract.** Automatically generated models may provide the key towards controlling the evolution of complex systems, form the basis for test generation and may be applied as monitors for running applications. However, the practicality of automata learning is currently largely preempted by its extremely high complexity and unrealistic frame conditions. By optimizing a standard learning method according to domain-specific structural properties, we are able to generate abstract models for complex reactive systems. The experiments conducted using an industry-level test environment on a recent version of a telephone switch illustrate the drastic effect of our optimizations on the learning efficiency. From a conceptual point of view, the developments can be seen as an instance of optimizing general learning procedures by capitalizing on specific application profiles.

## 1 Introduction

### 1.1 Motivation

The aim of our work is improving quality control for reactive systems as can be found e.g. in complex telecommunication solutions. A key factor for effective quality control is the availability of a specification of the intended behavior of a system or system component. In current practice, however, only rarely precise and reliable documentation of a system's behavior is produced during its development. Revisions and last minute changes invalidate design sketches, and while systems are updated in the maintenance cycle, often their implementation documentation is not. It is our experience that in the telecommunication area, revision cycle times are extremely short, making the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitate extensive testing effort. All this could be dramatically improved if it were possible to generate and then maintain appropriate reference models steering the testing effort and helping to evaluate the test results. In [6] it has been proposed to generate the models from previous system versions, by using learning techniques, and incorporating further knowledge in various ways. We call this general approach *moderated regular extrapolation*, which is tailored for a posteriori model construction and model updating during the system's life cycle. The general method includes many different theories and techniques [14].

In this paper we address the major bottlenecks of moderated regular extrapolation (cf. Sec. 1.3) – the *size* of the extrapolated models and the *performance* of the learning procedure – by application-specific optimization: Looking at the structural properties of the considered class of telecommunication systems allowed a significant advancement. This approach provides us with a key towards successfully exploiting automata learning, a currently mostly theoretical research area, in an industrial setting for the testing of telecommunication systems. Moreover it illustrates again that practical solutions can be achieved by ‘intelligent’ application-specific adaptation of general purpose solutions. In fact, we observed in our experiments that, even after eliminating all the ‘low hanging fruits’ resulting from prefix closedness and input determinism, the symmetry and partial order reductions still added a performance gain of almost an order of magnitude (cf. Sec. 3). As the impact of this optimization is non-linear, this factor will typically grow with the size of the extrapolated models.

Learning and testing are two wide areas of very active research — but with a rather small intersection so far, in particular wrt. practical application. At the theoretical level, a notable exception is the work of [13,4]. There, the problem of learning and refining system models is studied, e.g. with the goal of testing a given system for a specific property, or correcting a preliminary or invalidated model. In contrast, our paper focuses on enhancing the practicality of learning models to be able to capture real-life systems. This requires powerful optimizations of the concepts presented in [6,14].

## 1.2 Major Applications

Regression testing provides a particularly fruitful application scenario for using extrapolated models. Here, previous versions of a system are taken as the reference for the validation of future releases. By and large, new versions should provide everything the previous version did. I.e., if we compare the new with the old, there should not be too many essential differences. Thus, a model of the previous system version could serve as a useful reference to much of the expected system behavior, in particular if the model is abstract enough to focus on the essential functional aspects and not on details like, for instance, exact but irrelevant timing issues. Such a reference could be used for:

- Enhanced test result evaluation: Usually, success of a test is measured only via very few criteria. A good system model provides a much more thorough evaluation criterion for success of a test run.
- Improved error diagnosis: Related to the usage above, in case of a test error, a model might expose already very early some discrepancy to expected behavior, so that using a model will improve the ability to pinpoint an error.
- Test-suite evaluation and test generation: As soon as a model is generated, all the standard methods for test generation and test evaluation become applicable (see [5,2] for surveys about test generation methods).



### 1.3 Learning Finite Automata — Theory and Practice

Constructing an acceptor for an unknown regular language can be difficult. If the result is to be exact, and the only source of information are tests for membership in the considered language and a bound on the number of states of the minimal accepting deterministic finite automaton (DFA) for the language, the worst-case complexity is exponential [10] in the number of states of the acceptor. Additional information enables learning in polynomial time: The algorithm named  $L^*$  from [1] requires tests not only for membership, but also for equivalence of a tentative result with the language to be captured. The equivalence test enables the algorithm to always construct a correct acceptor in polynomial time. If the equivalence test is dropped, it is still possible to learn approximately in polynomial time, meaning that with high probability the learned automaton will accept a language very close to the given one [1].

We base our model construction procedure on  $L^*$ . To make this work in practice, there are several problems to solve. First of all, the worlds of automata learning and testing, in particular the testing of telecommunication systems, have to be matched. A telecommunication system cannot be readily seen as a finite automaton. It is a reactive, real-time system which receives and produces signals from large value domains. To arrive at a finite automaton, one has to abstract from timing issues, tags, identifiers and other data fields. Different from other situations where abstraction is applied, for instance in validation by formal methods, here it is of crucial importance to be able to reverse the abstraction. Learning is only feasible if one can check actively whether a given abstract sequence corresponds to (is an abstraction of) a concrete system behavior, and  $L^*$  relies heavily on being able to have such questions answered. I.e., in order to be able to resolve the *membership queries*, abstract sequences of symbols have to be retranslated into concrete stimuli sequences and fed to the system at hand. This problem has been solved with the help of a software enhancement of an existing, industrial test environment [11,12].

A major cause of the difficulties is the fact that  $L^*$  – as all its companions – does not take peculiarities of the application domain into account. Though a suitable abstraction of a telecommunication system yields indeed a finite automaton, it is a very special one: Its set of finite traces forms a prefix-closed language, every other symbol in the trace is an output determined by the inputs received by the system before, and, assuming correct behavior, the trace set is closed under rearrangements of independent events. It was the last property, which caused most problems when adapting the learning scenario. In fact, identifying independence and the concrete corresponding exploitation for the reduction of memberships queries both turned out to be hard. However, since physical testing, even if automated as in our case, is very time intensive, these optimizations are vital for practicality reasons.

In the following, Sec. 2 prepares the ground for our empirical studies and optimizations, Sec. 3 elaborates on the application-specific characteristics of the considered scenario, and presents the implemented optimizations together with their effects. Finally Sec. 4 draws some conclusions and hints at future work.

## 2 Basic Scenario

In this section, we set the stage for our application-specific optimizations: Sec. 2.1 describes the considered application scenario, Sec. 2.3 our application-specific elaborations of the underlying basic learning algorithm together with the major required adaptations, and Sec. 2.4 describes our experimentation scenario, which allows us to evaluate our technique in an industrial setting.

### 2.1 Application Domain: Regression Testing of Complex, Reactive Systems

The considered application domain is the functional regression testing of complex, reactive systems. Such systems consist of several subcomponents, either hardware or software, communicating with and affecting each other. Typical examples for this kind of systems are *Computer Telephony Integrated (CTI) systems*, like complex *Call Center solutions*, embedded systems, or web-based applications. To actually perform tests, we use a tool called *Integrated Test Environment (ITE)* [11,12] which has been applied in research and in industrial practice for different tasks.

One important property for regression testing is that results are reproducible, i.e., that results are not subject to accidental changes. In the context of telecommunication systems, one observes that a system’s reaction to a stimulus may consist of more than one output signal. Those outputs are produced with some delay, and in everyday application some of them may actually occur only after the next outside stimulus has been received by the system. Usually, functional testing does not care for exact timing. So the delays do not matter much. But it is very important to be able to match outputs correctly to the stimuli which provoked them. Thus, it is common practice to wait after each stimulus to collect all outputs. Most often, appropriate timeouts are applied to ensure that the system has produced all responses and settled in a “stable” state. If all tests are conducted in this way, for interpreting, storing and comparing test results, a telecommunication system can be viewed as an *I/O-automaton*: a device which reacts on inputs by producing outputs and possibly changing its internal state. It may also be assumed that the system is *input enabled*, i.e. that it accepts all inputs regardless of its internal state.

This is not all we need for our learning approach to work. First of all, we would like to look at the system as a propositional, that is finite, object. This means that we would like to have only finitely many components, input signals and output signals. W.r.t. the number of components, again we are helped by common testing practice. The complexity of the testing task necessitates a restriction to small configurations. So only a bounded number of addresses and component identifications will occur, and those can accordingly be represented by propositional symbols. Other components of signals can be abstracted away, even to the point where system responses to stimuli get deterministic – which is another important prerequisite for reproducible, unambiguously interpretable test results.

Summarizing, we adopted the following common assumptions and practices from real-life testing.

1. Distinction between stimuli and responses
2. Matching reactions to the stimuli that cause them
3. Restriction to finite installations
4. Abstraction to propositional signals

This leads to a view of a reactive/telecommunication system as a propositional, input-deterministic I/O-automaton.

**Definition 1.** An input/output automaton is a structure  $\mathcal{S} = (\Sigma, A_I, A_O, \rightarrow, s_0)$ , consisting of a finite, non-empty set  $\Sigma$  of states, finite sets  $A_I$  and  $A_O$  of input, resp. output, actions, a transition relation  $\rightarrow \subseteq \Sigma \times A_I \times A_O^* \times \Sigma$ , and a unique start state  $s_0$ . It is called input deterministic if at each state  $s$  there is at most one transition for each input starting from that state. It is input enabled if there is at least one transition for each input.

I/O automata according to this definition are not to be confused with the richer structures from [8]. As we are not concerned with automata algebra, we can use this simple form here<sup>1</sup>. As explained above, the system models we consider are all finite-state, input-deterministic I/O automata. Additionally, we can assume that they are input enabled.

## 2.2 L\*: A Basic Learning Algorithm

Angluin describes in [1] a learning algorithm for determining an initially unknown regular set exactly and efficiently. To achieve this aim, besides the alphabet  $A$  of the language two additional sources of information are needed: a *Membership Oracle* (MO), and an *Equivalence Oracle* (EO). A *Membership Oracle* answers the question whether a sequence  $\sigma$  is in the unknown regular set or not with **true** or **false**, whereas an *Equivalence Oracle* is able to answer the question whether a conjecture (an acceptor for a regular language) is equivalent to the unknown set with **true** or a counterexample.

The basic idea behind Angluin's algorithm is to systematically explore the system's behavior using the membership oracle and trying to build the transition table of a deterministic finite automaton with a minimal number of states. During the exploration, the algorithm maintains a set  $S$  of state-access strings, a set of strings  $E$  distinguishing the states found so far, and a finite function  $T$  mapping strings of  $(S \cup S \cdot A) \cdot E$  to  $\{0, 1\}$ , indicating results of membership tests. This function  $T$  is kept in the so-called *observation table*  $\mathcal{OT}$  which is the central data structure of the algorithm and from which the conjectures are read off. For a detailed discussion about L\* please refer to [1].

---

<sup>1</sup> Another common name for our type of machine is *Mealy Automaton*, only we permit a string of output symbols at each transition.

### 2.3 Application-Specific Adaptations to $L^*$

Before the basic  $L^*$  algorithm can be used in the considered scenario, some adaptations have to be made:

1. Representing I/O automata as ordinary automata, and
2. Practical implementation of a membership and an equivalence oracle

**Formal Adaptations** I/O automata differ from ordinary automata in that their edges are labelled with inputs and outputs instead of just one symbol. Obviously we could view a combination of an input and the corresponding output symbols as one single symbol. In our scenario this would result in a very large alphabet, in particular because outputs of an I/O automaton are sequences of elementary output symbols. Running  $L^*$  with such a large alphabet would be very inefficient. So we chose to split an edge labelled by a sequence of one input and several output symbols into a sequence of auxiliary states, connected by edges with exactly one symbol. In this way, we keep the alphabet and the observation table small (even though the number of states increases). Further reductions, which are possible because of the specific structure of the resulting automata, prohibit a negative impact of the state increase on the learning behavior. They are discussed in Sec. 3.

Given a system represented by an input-deterministic, input-enabled I/O automaton  $\mathcal{S}$ , our adaptation of  $L^*$  will learn a minimal DFA equivalent to  $\mathcal{DFA}(\mathcal{S})$ , the formally adapted version of  $\mathcal{S}$ . This equivalent DFA can be constructed rather straightforwardly: for each transition of  $\mathcal{S}$ , a set of corresponding transitions, with transient states in-between, will be created, where each transition is now labelled with a single alphabet symbol only. At each state (new or old), transitions for alphabet symbols not already labelling a transition exiting this states are introduced which end at an artificial error state.

Additionally the adequate realization of a membership and an equivalence oracle is required:

**Membership Oracle** Membership queries can be answered by testing the system we want to learn. This is not quite as simple as it sounds, mainly because the sequence to be tested is an abstract, propositional string, and the system on the other hand is a physical entity whose interface follows a real-time protocol for the exchange of digital (non-propositional) data. For processing a membership query a so-called test graph for the *ITE* has to be constructed, which describes the sequence of stimuli and expected responses, to actually being able to execute membership queries. In essence, the *ITE* bridges the gap between the abstract model and the concrete system.

Although the tests are automatically generated, membership queries are very expensive: The automatic execution of a single test took approximately 1.5 minutes during our experimentation because of timeouts of the system that have to be considered. Note that the long execution time is due to the generous timeouts that are specified for telecommunication systems. But as in almost every

real-time system timeouts are of central importance, it is expected, that the test execution time affects the overall performance of such approaches in general. Therefore it is very important to keep the number of such membership queries low.

**Equivalence Oracle** For a black-box system there is obviously no reliable equivalence check – one can never be sure whether the whole behavior spectrum of a system has been explored. But there are approximations which cover the majority of systems pretty well. The basic idea is to scan the system in the vicinity of the explored part for discrepancies to the expected behavior. One particularly good approximation is achieved by performing a systematic conformance test in the spirit of [3] which looks at source and goal of every single transition. Another possibility lies in checking consistency within a fixed lookahead from all states. In the limit, by increasing the lookahead, this will rule out any uncertainty, though at the price of exponentially increasing complexity.

In our experiments, it was sufficient to use the next system outputs as a lookahead for obtaining satisfactory results. We expect, however, that this will have to be extended for capturing more complex cases. We additionally foresee the integration of expert knowledge into the learning scenario, e.g. by checking temporal-logic constraints on tentative results (see also [6]).

## 2.4 Basic Learning in Practice

With our framework established so far, we are able to learn models for finite installations of (a part of) a complex call center solution (cf. [11], [12, Fig. 4]). A midrange telephone switch is connected to the public telephone network and acts as a 'normal' telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PC's.

Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls). Moreover the applications can be seen as logical devices, which cannot only be used as a phone, but form a compound with physical devices, so that they can complement each other, e.g. an application can initiate a call on a physical phone based on a user's personal address book.

The technical realization of the necessary interface to this setup is provided by the *ITE*, in particular the *Test Coordinator*, which controls in particular two different test tools, i.e. a test tool to control the telephone switch (*Husim*) and one for the clients (*Rational Robot*). This is described in more detail in [12].

We have carried out our experiments on four finite installations, each consisting of the telephone switch connected to a number of telephones (called "physical devices"). The telephones were restricted differently in their behavior, ranging from simple on-hook ( $\uparrow$ ) and off-hook ( $\downarrow$ ) actions of the receiver to actually performing calls ( $\rightarrow$ ). The four installations are listed below. Note that we have two versions of the first three scenarios, depending on the observability of the signal (*hookswitch*).

- $S_1$ : 1 physical device ( $A$ ),  $A_I = \{A \uparrow, A \downarrow\}$ ,  
 $A_O = \{initiated_A, cleared_A, [hookswitch_A]\}$ .
- $S_2$ : 2 physical devices ( $A, B$ ),  $A_I = \{A \uparrow, A \downarrow, B \uparrow, B \downarrow\}$ ,  
 $A_O = \{initiated_{\{A,B\}}, cleared_{\{A,B\}}, [hookswitch_{\{A,B\}}]\}$ .
- $S_3$ : 3 physical devices ( $A, B, C$ ),  $A_I = \{A \uparrow, A \downarrow, B \uparrow, B \downarrow, C \uparrow, C \downarrow\}$ ,  
 $A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, [hookswitch_{\{A,B,C\}}]\}$ .
- $S_4$ : 3 physical devices ( $A, B, C$ ),  $A_I = \{A \uparrow, A \downarrow, A \rightarrow B, B \uparrow, B \downarrow, C \uparrow, C \downarrow\}$ ,  
 $A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, originated_{A \rightarrow B}, established_B\}$

The output events indicate which actions the telephone switch has performed on the particular input. For instance, *initiated<sub>A</sub>* indicates that the switch has noticed that the receiver of device  $A$  is off the hook, whereas *initiated<sub>{A,B}</sub>* denotes that we have both events *initiated<sub>A</sub>* and *initiated<sub>B</sub>*. With *originated<sub>A→B</sub>* the switch reports that it has forwarded the call request from  $A$  to  $B$ . In a more general setting, more system responses would lead to more transient states in the learned DFA.

The third column of Tab. 1 in Sec. 3.2 points out impressively that for learning even relatively small scenarios the number of membership queries is quite large. Therefore it is clearly unavoidable to reduce the number of membership queries drastically. In fact, we would have had difficulties to compute these figures in time for the publication without using some reductions. To solve this problem additional filters have been added to the connection from  $L^*$  to the two oracles. These filters reduce the number of queries to the oracles by answering queries themselves in cases where the answer can be deduced from previous queries. In our experiments, we used properties like determinism, prefix closure and independence of events for filter construction. The next section formally develops this novel way of applying profile-dependent optimization to the learning process, which we consider a key for the practicality of automata learning.

### 3 Optimized Learning

#### 3.1 Theory

This section describes several optimizations given through rules which can be used to filter the membership queries. Common to all rules is that they are able to answer the membership queries from the already accumulated knowledge about the system. They will be presented in the form **Condition**  $\Rightarrow \dots \{\mathbf{true}, \mathbf{false}\}$ . **Condition** refers to the  $\mathcal{OT}$  in order to find out whether there is an entry in the table stating that some string  $\sigma$  is already known to be member of the set to be learned or not, i.e. whether we are interested in  $T(\sigma)$ .

The filter rules that will be presented are based on several properties of (special classes of) reactive systems. They are to some extent orthogonal to each other and could be applied independently. This potentially increases the usability both of our current concrete approach and of the general scenario. Given another learning problem, a possible way to adapt the learning procedure is to identify a profile of the new scenario in terms of specific properties and to optimize the learning process by means of the corresponding filter rules.

**Prefix-Closure:** In testing a reactive system, the observations made are finite prefixes of the potentially infinite sequences of inputs and outputs. Any error (i.e., non-acceptance of a string) implies that also no continuation of the string will be observed. Other than in general regular languages, there is no switching from non-accepting to accepting. Correspondingly, the DFAs resulting from translating an I/O automaton have just one sink as a non-accepting state. The language we want to learn is thus *prefix closed*. This property directly gives rise to the rule Filter 1, where each prefix of an entry in  $\mathcal{OT}$ , that has already been rated **true** (i.e. is a member of the target language), will be evaluated with **true** as well.

**Filter 1 (Positive Prefix)**  $\exists \sigma' \in A^*. T(\sigma; \sigma') = 1 \implies MO(\sigma) = \text{true}$

The contraposition states that once we have found a sequence  $\sigma$  that is rated **false**, all continuations have also to be rated **false**. This yields the rule Filter 2.

**Filter 2 (Negative Prefix)**  $\exists \sigma' \in \text{prefix}(\sigma). T(\sigma') = 0 \implies MO(\sigma) = \text{false}$

Together, these two rules capture all instances of membership queries which are avoidable when learning prefix-closed languages. Already these rather simple rules have an impressive impact on the number of membership queries as will be shown in Tab. 1.

**Input Determinism** Input determinism ensures that the system to be tested always produces the same outputs on any given sequence of inputs (cf. Definition 1). This implies that replacing just one output symbol in a word of an input-deterministic language cannot yield another word of this language.

**Proposition 1.** *Let  $\mathcal{S}$  be an input-deterministic I/O automaton and let furthermore  $\sigma \in \mathcal{L}(\mathcal{DFA}(\mathcal{S}))$ . Then the following holds.*

1. *If there exists a decomposition of  $\sigma = \sigma'; x; \sigma''$  with  $x \in A_O$ , then  $\forall y \in A \setminus \{x\}. \sigma'; y; \sigma'' \notin \mathcal{L}(\mathcal{S})$ .*
2. *If there exists a decomposition of  $\sigma = \sigma'; a; \sigma''$  with  $a \in A_I$ , then  $\forall x \in A_O. \sigma'; x; \sigma'' \notin \mathcal{L}(\mathcal{S})$ .*

The first property says that each single output event is determined by the previous inputs. It may be emphasized that this property is of crucial importance for learning reactive systems, as a test environment has no direct control over the outputs of a system. If the outputs were not determined by the inputs, there would be no way to steer the learning procedure to exhaustively explore the system under consideration.

The second property reflects the fact that the number of output events in a given situation is determined, and that we wait with the next stimulus until the system has produced all its responses. This is useful but not as indispensable as the first property. The corresponding filter rules are straightforward:

**Filter 3 (Input Determinism)**  $\exists x \in A_O, y \in A, \sigma', \sigma'' \in A^*. \sigma = \sigma'; x; \sigma'' \wedge T(\sigma'; y; \sigma'') = 1 \wedge x \neq y \implies MO(\sigma) = \text{false}$

**Filter 4 (Output Completion)**  $\exists a \in A_I, x \in A_O. \sigma'; x \in \text{prefix}(\sigma) \wedge T(\sigma'; a) = 1 \implies MO(\sigma) = \text{false}$

**Independence of Events:** Reactive systems exhibit very often a high degree of parallelism. Moreover in telecommunication systems several different components of one type, here e.g. telephones, can often be regarded as generic so that they are interchangeable. This can lead to symmetries, e.g. a device  $A$  behaves like device  $B$ .

*Partial order reduction* methods for communicating processes [9,15] deal with this kind of phenomena. Normally these methods can help avoiding to examine all possible interleavings among processes. However, as we will illustrate here, these methods can also be used to avoid unnecessary membership queries using the following intuition:

**Independence** If device  $A$  can perform a certain action and afterwards device  $B$  can perform another, and those two actions are independent, then they can be performed in different order (i.e., device  $B$  starts) as well.

**Symmetries** If we have seen that device  $A$  can perform a certain action we know that the other (similar) devices can perform it as well.

This implies that the membership question for a sequence  $\sigma$  can be answered with **true** if an equivalent sequence  $\sigma'$  is already in  $\mathcal{OT}$ , where two sequences are equivalent if they are equal up to partial order and symmetries. Using  $C_{po,sym}$  to denote the *partial order completion* with respect to *symmetries*, i.e. the set of all equivalent sequences up to the given partial order and symmetry, we obtain the following filter rule.

**Filter 5 (Partial Order)**  $\exists \sigma' \in C_{po,sym}(\sigma). T(\sigma') = 1 \implies MO(\sigma) = \text{true}$

Whereas the general principle of Filter 5 is quite generic, the realization of  $C_{po,sym}$  strongly depends on the concrete application domain. However, the required process may always follow the same pattern:

1. An expert specifies an application-specific **independence relation**, e.g. *Two independent calls can be shuffled in any order.*
2. During an *abstraction* step the concrete component identifiers will be replaced by generic place holders with respect to an appropriate symmetry.
3.  $\sigma$  is inspected if it contains independent subparts with respect to the independence relation.
4. All possible re-orderings will be computed.
5. The generic place holders will be *concretized* again (with all possible assignments due to symmetries).



**Table 1.** Number of Membership Queries

Scenario	States	no filter	Fil. 1 & 2	Factor	Fil. 3 & 4	Factor	Fil. 5	Factor	Tot. Factor
$S_1$	4	108	30	3.6	15	2.0	14	1.1	7.7
$S'_1$	8	672	181	3.7	31	5.8	30	1.0	22.4
$S_2$	12	2,431	593	4.1	218	2.7	97	2.2	25.1
$S'_2$	28	15,425	4,080	3.8	355	11.5	144	2.5	107.1
$S_3$	32	19,426	4,891	4.0	1,217	4.0	206	5.9	94.3
$S'_3$	80	132,340	36,374	3.6	2,007	18.1	288	7.0	459.5
$S_4$	78	132,300	29,307	4.5	3,851	7.6	1,606	2.4	81.1

In the following we will discuss how  $C_{po,sym}$  works in the context of telephony systems by presenting a simple example. Consider a test run, where device  $A$  calls device  $B$  and after that device  $B$  answers the call so that it will be established. As a last action, device  $C$  also picks up the receiver. This action of  $C$  is independent from the previous ones, as device  $C$  is not involved in the connection between  $A$  and  $B$ . Therefore it does not matter at which time  $C$  performs its action.

In a first step the dependencies between the involved devices will be computed and this leads to two independent subsequences: one containing the call between device  $A$  and  $B$ , and the other contains the `hookoff` action by device  $C$  together with the responses of the switch to it. After that by an anonymization step every concrete device will be replaced by an *actor name* ( $\alpha_1, \alpha_2, \alpha_3$ ), i.e. every device will be treated as a generic device. This reflects the fact, that if we have observed once that e.g. device  $A$  can perform a certain action every other equivalent device can perform it as well. For the further processing it is of intrinsic importance that this abstraction step is reversible. In the next step, from these two independent subsequences, which describe a partial order, a directed acyclic graph representing all possible interleavings is computed. Finally, all concrete sequences that can occur through binding the actors again to the concrete devices (without using a device identifier twice) are determined, in order to e.g. treat the sequence where  $C$  is in connection with  $A$  and  $B$  picks up its receiver at some point as equivalent to the original one. More precisely we perform a concretization step where the abstract actions, referencing the different actors  $\alpha_i$ , together with the set of available concrete devices in the considered setting ( $Dev = \{A, B, \dots\}$ ), will be transformed into concrete actions, suitable for the further processing within the learning algorithm.

### 3.2 Practice

In the following, we discuss the impact of the filters to the scenarios defined in Sec. 2.4. We have run the learning process with all available filters applied in a cumulative way, i.e., when using Filters 3 and 4, also Filters 1 and 2 were used. The results are summarized in Tab. 1. The “Factor” columns in the table provide the additional reduction factor in the number of membership queries achieved by successively adding new filters.

As one can see we were able to reduce the total number of membership queries in all scenarios drastically. In the most drastic case ( $S'_3$ ), we only needed a fraction of a percent of the number of membership queries that the basic  $L^*$  would have needed. In fact, learning the corresponding automaton without any optimizations would have taken about 4.5 months of computation time.

The prefix reduction (Filters 1 and 2) has a similar impact in all considered scenarios. This seems to indicate that it does not depend very much on the nature of the example and on its number of states.

The other two reductions (input determinism and partial order) vary much more in their effectiveness. Note that the saving factor increases with the number of states. Shifting attention to the number of outputs and the lengths of output sequences between inputs, these seem to have a particular high impact on the effects of the Filters 3 and 4. This can be seen by comparing the scenarios  $S_i$  with their counterparts  $S'_i$ . In these counterparts an additional output event is modelled, the hookswitch event, which occurs very frequently, namely after each of the permitted inputs.

One would expect that the impact of Filter 5, which covers the partial-order aspects, will increase with the level of independency. And indeed we find this conjecture confirmed by the experiments.  $S_1$  has only one actor so that there is no independence, which results in a factor of 1. As the number of independent devices increases, the saving factor increases as well, see for instance the figures for  $S_2$  and  $S_3$ . It is worth noting that the number of states does not seem to have any noticeable impact on the effectiveness of this filter, as the reduction factor more or less remains equal when switching from  $S_i$  and  $S'_i$ . Compared to  $S_3$ , the saving factor in  $S_4$  decreases. This is due to the fact that an action has been added in  $S_4$  that can establish dependencies between two devices, namely the initiation of a call.

## 4 Conclusion

We have demonstrated that with the right modifications to the learning procedure, and the right environment for observing system, system models may in fact be learned in practice. Modifying the learning algorithm by filtering out unnecessary queries enabled us to perform the experiments easily, and in general it provides a flexible approach which permits fast adaptations to different application domains. We are convinced that, at least for certain application scenarios, automata learning will turn out to be a powerful means for quality assurance and the control of system evolution.

For fixed settings, modifications to the algorithm itself should be made. For instance, it would not be hard to tailor  $L^*$  to cope better with prefix-closed I/O-automata. However we conjecture that even after such a tailoring the presented filters will have a significant impact. A direct adaptation of the learning algorithm for capturing partial order and symmetry is far more involved. E.g., one is tempted to learn a reduced language whose partial-order closure is the full set of strings instead of the full language. However, it is known that the set

of strings reduced wrt. a partial order need not be regular [9]. Thus there is still enormous potential, both for further optimizations and for basic research.

## 5 Acknowledgement

We thank Tiziana Margaria for fruitful discussions, as well as the anonymous referees for many helpful comments on early versions of this paper.

## References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.
2. E. Brinksma, and J. Tretmans. Testing transition systems: An annotated bibliography in *Proc. of MOVEP'2k*, pages 44–50, 2000.
3. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
4. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. TACAS '02*, LNCS 2280, pages 357–370. Springer Verlag, 2002.
5. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. of the IEEE*, 84:1090–1126, 1996.
6. A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In *Proc. of FASE '02*, LNCS 2306, pages 80–95. Springer Verlag, 2002.
7. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
8. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.
9. A. Mazurkiewicz. Trace theory. In *Petri Nets, Applications and Relationship to other Models of Concurrency*, LNCS 255, pages 279–324. Springer Verlag, 1987.
10. E.F. Moore. Gedanken-experiments on sequential machines. *Annals of Mathematics Studies (34)*, *Automata Studies*, pages 129–153, 1956.
11. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In *Proc. of FASE '01*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
12. O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, H. Ide, and W. Goerigk. Automated regression testing of CTI-systems. In *Proc. IEEE ETW' 01*, pages 51–57, IEEE Press, 2001.
13. D. Peled, M.Y. Vardi, and M. Yannakakis. Black box checking. In *Proc. FORTE/PSTV '99*, pages 225–240. Kluwer Academic Publishers, 1999.
14. B. Steffen, and H. Hungar. Behavior-Based Model Construction. In *Proc. of VMCAI '02*, LNCS 2575, pages 5–19. Springer Verlag, 2002.
15. A. Valmari. On-the-fly verification with stubborn sets. In *Proc. of CAV '93*, LNCS 697, pages 397–408. Springer Verlag, 1993.

# Model Checking Conformance with Scenario-Based Specifications \*

Marcelo Glusman and Shmuel Katz

Department of Computer Science  
The Technion, Haifa 32000, Israel  
{marce, katz}@cs.technion.ac.il

**Abstract.** Specifications that describe typical scenarios of operations have become common for software applications, using, for example, use-cases of UML. For a system to conform with such a specification, every execution sequence must be equivalent to one in which the specified scenarios occur sequentially, where we consider computations to be equivalent if they only differ in that independent operations may occur in a different order.

A general framework is presented to check the conformance of systems with such specifications using model checking. Given a model and additional information including a description of the scenarios and of the operations' independence, an augmented model using a transducer and temporal logic assertions for it are automatically defined and model checked. In the augmentation, a small *window* with part of the history of operations is added to the state variables. New transitions are defined that exchange the order of independent operations, and that identify and remove completed scenarios. If the model checker proves all the generated assertions, every computation is equivalent to some sequence of the specified scenarios. A new technique is presented that allows proving equivalence with a small fixed-size window in the presence of unbounded out-of-order of operations from unrelated scenarios. This key technique is based on the *prediction* of events, and the use of *anti-events* to guarantee that predicted events will actually occur. A prototype implementation based on Cadence SMV is described.

**Keywords:** Model checking, scenario, computation equivalence, transducer, anti-event

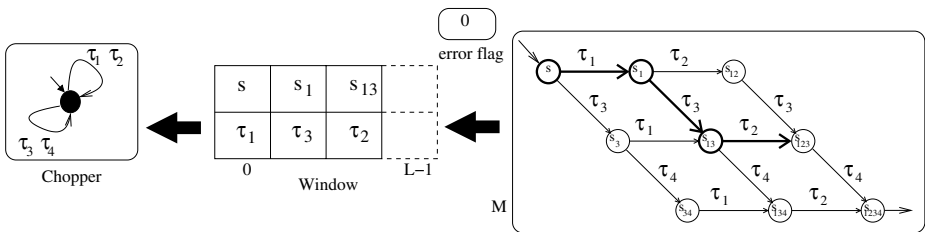
## 1 Introduction

It has become common to describe systems through a series of typical, canonic, scenarios of behavior, either through *use-cases* of UML or other formalisms. System computations in which such scenarios occur in sequence are called *convenient*. We consider two computations that differ only in the order in which independent operations are executed, as *equivalent* with respect to a scenario-based specification. A system conforms with such a specification if every computation is equivalent to a convenient one. The specifications of *serializability* of concurrency control algorithms for distributed databases and, in the area of hardware design, the *sequential consistency* of shared memory protocols have a similar structure.

---

\* This work was partially supported by the Fund for the Support of Research at the Technion.

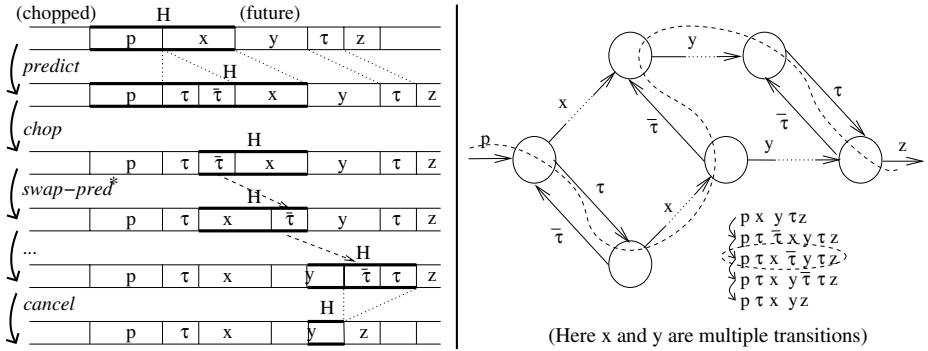
Given a fair transition system  $M$ , we build a transducer  $M'$  (See Fig.1) by composing  $M$  with a bounded *window* (a history queue)  $H$  of fixed length  $L$ , and an  $\omega$ -automaton  $C$  (which we also call the *chopper*) that reads its input from  $H$  and accepts only the desired scenarios. In addition,  $M'$  has an *error* flag, initially false. When a (non-idling) transition from  $M$  is taken, if the history is not full and the error flag is false, the current state-transition pair is enqueued in  $H$ . Otherwise, the error flag is set to true, and never changed thereafter. When the error flag is true, there is no further interaction between  $M$  and  $H$ . Additional transitions of  $M'$  are *swaps* of consecutive elements of  $H$  (according to the user-defined conditional independence) and *chops* of prefixes of the window, that remove any desired scenario recognized by  $C$ . A run of  $M$  is defined as convenient if it belongs to the language of  $C$ , i.e., is composed of desired scenarios. Any infinite sequence built by concatenation of the chopped state-transition pairs is therefore a convenient run. We aim to prove that for *every* computation  $g$  of  $M$  there is a convenient



computation  $c$  such that  $c \approx g$ . The transducer augments  $M$  without interfering with its operation. Therefore, for every computation  $g$  of  $M$ , there is some computation  $g'$  of  $M'$  such that  $g$  is the projection of  $g'$  to  $M$ 's variables and transitions. If there is such a  $g'$  in which  $M$ 's transitions are interleaved with swaps in  $H$  and transitions of  $C$  (that dequeue elements from  $H$ ), so that  $H$  is never full when  $M$  makes a transition, then the error flag will never be raised in  $g'$ .

A bounded transducer cannot deal directly with events that may be delayed without bound, e.g., if they can be preceded by any number of independent events (“unbounded out-of-order”). Such a situation may arise in many concurrent systems, where the fairness constraints on the computations do not set any bound on how long a transition can be delayed before it is taken. If the window were *not* bounded, the transducer could wait until the needed event occurs, then swap it “back in time” until it reaches its place in a convenient prefix. A bounded window will necessarily become full in some computations, and the *error* flag will be raised before the needed event occurs.

To overcome unbounded out-of-order, when part of a scenario has been identified a *prediction* can be made that a currently enabled event will eventually occur. In our novel prediction mechanism, the transducer then creates and inserts an event/anti-event pair after the existing part of a scenario being built in the window. After the predicted event is chopped together with the rest of the scenario, the anti-event remains in the window to keep track of the prediction. The anti-event can be swapped with the following events in the window, only if its corresponding event could have been swapped in the opposite direction. When the anti-event later meets the actual occurrence of its predicted event, they cancel out and disappear (see Fig. 2).<sup>1</sup> For this method to work, we need to make sure



**Fig. 2.** Predicting  $\tau$  to prove  $pxy\tau z \equiv_{sw} p\tau xyz$ .

every event/anti-event pair is inserted after any previously predicted event and before its corresponding anti-event. This method is analogous to the use of prophecy variables, but is easier to implement and understand.

The generated LTL and CTL proof obligations over the extended model verify that for every computation  $g$  of  $M$  there is a computation  $c$  of  $M'$  that “simulates”  $g$ , while avoiding raising the *error* flag, fulfilling all the predictions, and in which no event remains forever in the window. They also require that it is always possible to postpone the initiation of new scenarios until the pending ones are completed. Together with a

<sup>1</sup> A nice analogy can be drawn between the behavior of our anti-events (and events) and that of particles (resp., antiparticles) in modern physics, where a positron can be seen as an electron travelling back in time.

restriction on the independence relation, needed to preserve fairness, the correctness of all these formulas for  $M'$  guarantees that  $M$  conforms with the scenario-based specification given by  $C$ .

To demonstrate the feasibility of our approach, we built a prototype tool (CNV) that implements our method. A description of the problem in CNV's language is used to generate automatically the transducer and the related formulas, both in the language of the model checker Cadence SMV [15], which is then used to verify them. In CNV, additional facilities (beyond prediction) are provided to reduce the size of the window, such as a means for defining a small abstraction of the state information that suffices to support the transducer's operation.

**Related work:** The Convenient Computations proof method [4, 5, 11] is based on showing that every computation of a system can be reduced or is equivalent to one in a specified subset called the *convenient* computations. These computations are easier to verify for properties that should hold for the entire system. In a proof, one must define which computations are convenient, prove that they satisfy the property of interest, and then extend this result by showing that a property-preserving reduction exists from every computation to a convenient one. This general approach is useful both for (potentially infinite state) systems with scenario-based specifications, as is done here, and for dividing inductive property verification efforts into several subproblems. Manual proofs following this approach appeared before in [9, 11]. In [4], a proof environment based on a general-purpose theorem prover was presented. As usual in a deductive theorem-proving tool, the proof environment is wide in its range of applicability but shallow in the degree of automation it provides. However, it formalizes the proof method's definitions, and encapsulates key lemmas needed in a theorem proving setting.

The work presented here also differs from partial order reductions in model checking [2, 7, 17, 18], that exploit similar notions of independence to reduce the size of the model being checked. In those works there is no separate description of the desired convenient computations. Moreover, only restricted forms of independence can be used, and many computations that are not convenient are left in the reduced version, because the reduction is conservative, and is done on-the-fly based on little context information. Thus those works use some related ideas to solve a completely different problem.

Transduction methods have been used in a theoretical deductive setting for the verification of partial-order refinements, and in particular sequential consistency of lazy caching, in [8]. There, the transducer holds a pomset of unbounded size, and the independence relation among operations is fixed. Finite-state serializers [3, 6] have been used to verify Sequential Consistency for certain classes of shared memory protocols by model checking. These can be seen as special cases of the transduction idea, optimized and tailored to support verification of that specific property.

This paper is organized as follows: In Section 2 we provide basic definitions about the reductions and equivalence relations we want to prove. In Section 3 we show how to augment a given transition system to enable model checking equivalence of every computation to a convenient one. In Section 4 we describe a prototype tool that implements the techniques described in the previous sections, and apply it to a simple yet paradigmatic example. Section 5 concludes with a discussion.

## 2 Theoretical Background

### 2.1 Computation Model, Conditional Independence

The following definition of Fair Transition Systems (FTS) is based on [13].

**Definition:**  $M = \langle V, \Theta, \mathcal{T}, \mathcal{J} \rangle$  is an FTS iff: (i)  $V$  is a finite set of *system variables*. Let  $\Sigma$  denote the set of assignments to variables in  $V$ , also called *states*. (ii)  $\Theta$  is an assertion on variables from  $V$  (characterizing the *initial states*). (iii)  $\mathcal{T}$  is a finite set of *transitions*. Each  $\tau \in \mathcal{T}$  is a function  $\tau : \Sigma \rightarrow 2^\Sigma$ . We say that  $\tau$  is *enabled* on the state  $s$  if  $\tau(s) \neq \emptyset$ , which we denote  $en(s, \tau)$ . We require that  $\mathcal{T}$  includes an *idling* transition  $\iota$ , such that for every state  $s$ ,  $\iota(s) = \{s\}$ . (iv)  $\mathcal{J} \subseteq \mathcal{T} \setminus \{\iota\}$  is a set of transitions called *just* or *weakly fair*.

A *run* (or *execution sequence*)  $\sigma$  of  $M$  is an infinite sequence of state-transition pairs:  $\sigma \in (\Sigma \times \mathcal{T})^\omega$ , i.e.,  $\sigma : (s_0, \tau_0), (s_1, \tau_1), \dots$ , such that  $s_0$  satisfies  $\Theta$  (“initiality”), and  $\forall i \in \mathbb{N} : s_{i+1} \in \tau_i(s_i)$  (“consecution”). A transition  $\tau$  is *enabled at position  $i$*  in  $\sigma$  if it is enabled on  $s_i$ , and it is *taken* at that position if  $\tau = \tau_i$ . A *computation* is a run such that for each  $\tau \in \mathcal{J}$  it is not the case that  $\tau$  is continually enabled beyond some position  $j$  in  $\sigma$  but not taken beyond  $j$  (“justice”). We say that a state  $s$  is *quiescent* iff every transition  $\tau \in \mathcal{J}$  is disabled in  $s$ . A computation  $\sigma$  is *quiescent at position  $i$*  iff  $s_i$  is quiescent, and  $\tau_i = \iota$ .

Given a set  $S \subseteq \Sigma$  and a transition  $\tau$ , let  $\tau(S) = \bigcup_{s \in S} \tau(s)$ .

**Definition:** Two transitions  $\tau_1, \tau_2$  are *conditionally independent* in state  $s$  – denoted by  $CondIndep(s, \tau_1, \tau_2)$  – iff  $\tau_1 \neq \tau_2 \wedge \tau_2(\tau_1(s)) = \tau_1(\tau_2(s))$

In the rest of this paper we assume that all transitions of the system being analyzed are deterministic (i.e., their range has only empty or singleton sets), and if  $\tau(s) = \{s'\}$  we refer to  $s'$  as  $\tau(s)$ . Under this assumption, our definition of conditional independence coincides with the functional independence defined in [4], and with the restriction imposed on acceptable *collapses* in [10], for pairs of transitions that actually occur in consecution (which are candidates to be swapped).

### 2.2 Swap Equivalence

In the sequel we consider a user-defined conditional independence relation  $I \subseteq (\Sigma \times \mathcal{T} \times \mathcal{T})$  that is a subset of the full conditional independence  $CondIndep$ <sup>2</sup>, and such that  $\forall s \in \Sigma \ \forall \tau \in \mathcal{T} : I(s, \tau, \iota) \wedge I(s, \iota, \tau)$ . We define swap equivalence, following [4]. Note that for runs that share a suffix, the justice requirements defined by  $\mathcal{J}$  hold equally. Let  $\sigma = (s_0, \tau_0), (s_1, \tau_1), \dots$  be a computation of  $M$ , such that for some  $i \in \mathbb{N}$ ,  $I(s_i, \tau_i, \tau_{i+1})$  holds, and thus  $CondIndep(s_i, \tau_i, \tau_{i+1})$ . Then, the sequence  $\sigma' = (s_0, \tau_0), \dots, (s_i, \tau_{i+1}), (\tau_{i+1}(s_i), \tau_i), (s_{i+2}, \tau_{i+2}), \dots$  is also a legal computation of  $M$ , and we say that  $\sigma$  and  $\sigma'$  are *one-swap-equivalent* ( $\sigma \equiv_{1sw} \sigma'$ ). The *swap-equivalence* ( $\equiv_{sw}$ ) relation is the reflexive-transitive closure of one-swap-equivalence. The definitions of  $\equiv_{1sw}$  and  $\equiv_{sw}$  can also be applied to finite prefixes.

Swap-equivalence is the “conditional trace equivalence” (for finite traces) of [10]. If the independence relation  $I$  is fixed (i.e.,  $\forall s, t \in \Sigma, \forall \tau_1, \tau_2 \in \mathcal{T} : I(s, \tau_1, \tau_2) = I(t, \tau_1, \tau_2)$ ), swap-equivalence coincides with trace equivalence [14].

<sup>2</sup> we refer to sets and their characteristic predicates interchangeably

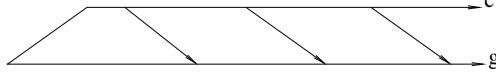


### 2.3 Conditional Trace Equivalence

Let  $\sigma^{\uparrow l} \in (\Sigma \times \mathcal{T})^l$  denote  $\sigma$ 's prefix of length  $l$ . For infinite runs  $g, c$ :

**Definition:**  $c \sqsubseteq g$  iff  $\forall m \in \mathbb{N} \exists h \in (\Sigma \times \mathcal{T})^\omega : (c^{\uparrow m} = h^{\uparrow m}) \wedge h \equiv_{sw} g$

The relation  $\sqsubseteq$  also appeared in [11, 12], but exploiting only fixed independence among transitions. If  $c \sqsubseteq g$ , there are finite execution paths from every state in  $c$  to one in  $g$  (See Fig.3) but  $c$  and  $g$  may not converge into a shared state.



**Fig. 3.**  $c \sqsubseteq g$ . (Converging paths represent swap-equivalent runs)

**Definition:** Two infinite runs  $g$  and  $c$  are *conditional trace equivalent* ( $g \approx c$ ) iff  $c \sqsubseteq g$  and  $g \sqsubseteq c$ .

### 3 Proving Equivalence by Model Checking

We define the transducer described in Section 1 as  $M' = (V', \Theta', T', \mathcal{J}')$ , where

(i)  $V'$  includes the variables in  $V$ , the *error* flag, a queue that can hold at most  $L$  state-transition pairs (the window), the state variables  $V_c$  of the chopper, and an index into  $H$ . Let  $\Sigma_c$  denote the possible states of  $C$ . The set of anti-transitions is:

$\overline{\mathcal{T}} = \{\bar{\tau} : \Sigma \rightarrow 2^\Sigma \mid \exists \tau \in \mathcal{T} \cdot \forall s_1, s_2 \in \Sigma, s_1 \in \bar{\tau}(s_2) \text{ iff } s_2 = \tau(s_1)\}$ . The set of transitions that can be recorded in the window is  $\mathcal{T}_h = \mathcal{T} \cup \overline{\mathcal{T}} \setminus \{\iota, \bar{\iota}\}$ . Let  $A^{0..L} = \cup_{i=0}^L A^i$ . The set of states of  $M'$  (valuations of  $V'$ ) is  $\Sigma' = \Sigma \times \{0, 1\} \times (\Sigma \times \mathcal{T}_h)^{0..L} \times \Sigma_c \times 0..L$ .  $M'$  is in state  $s' \in \Sigma' = (s, b, h, s_c, np)$  iff its  $M$  component is in the state  $s$ , the error flag has the boolean value  $b$ , the window's contents is  $h$ , the chopper is in the state  $s_c$  and the earliest point in the window where a prediction can be introduced is  $np$ . Let the predicate  $ep(h, np)$  hold iff  $np$  points to the place in the window right after the last predicted event.

(ii)  $\Theta' = \{(s_0, false, \epsilon, s_{c_0}, 0) \mid s_0 \models \Theta \wedge s_{c_0} \models \Theta_c\}$ , where  $\Theta_c$  characterizes the initial states of the chopper.

(iii)  $\mathcal{J}' = \{\tau' \mid \tau \in \mathcal{J}\}$

(iv)  $T'$  has the following transitions:

- $\tau'$ : Simulating transitions of  $M$ . For every  $\tau \in \mathcal{T}$ :  
 $\tau'((s, b, h, s_c, np)) = (\text{if } \tau = \iota \text{ then } \{(s, b, h, s_c, np)\} \text{ else if } (b = true \vee |h| = L) \text{ then } \{(\tau(s), true, h, s_c, np)\} \text{ else } \{(\tau(s), false, h \cdot (s, \tau), s_c, np)\})$ .
- *swap*: Swapping consecutive events in the window.  
 $swap((s, b, h_1, s_c, np)) = \{(s, b, h_2, s_c, np') \mid h_2 \equiv_{1sw} h_1 \wedge ep(h_2, np')\}$ .
- *chop*: Removing convenient prefixes from the window.  $chop((s, b, h_1, s_{c_1}), np) = \{(s, b, h_2, s_{c_2}, np') \mid \exists p \in (\Sigma \times \mathcal{T}_h)^{0..L}. h_1 = p \cdot h_2 \wedge s_{c_1} \xrightarrow{p} s_{c_2} \wedge ep(h_2, np')\}$ .<sup>3</sup>

<sup>3</sup>  $s_{c_1} \xrightarrow{p} s_{c_2}$  means that  $C$  can move from  $s_{c_1}$  to  $s_{c_2}$  by reading  $p$ .

- *predict*: Inserting an event/anti-event pair.  

$$\text{predict}((s, b, h, s_c, np)) = \{(s, b, p \cdot (s_1, \tau) \cdot (\tau(s_1), \bar{\tau}) \cdot q, s_c, np') \mid h = p \cdot q \wedge (q = \epsilon \rightarrow s_1 = s) \wedge (q \neq \epsilon \rightarrow (s_1 \text{ is the first state of } q)) \wedge p \in (\Sigma \times \mathcal{T})^* \wedge |p| \geq np \wedge np' = |p| + 1\}$$
- *swap-pred*: Swapping an anti-transition with an unrelated following transition.  

$$((s, b, h, s_c, np)) = \{(s, b, p \cdot (\tau(s_1), \alpha) \cdot (\alpha(\tau(s_1)), \bar{\tau}) \cdot q, s_c, np) \mid h = p \cdot (\tau(s_1), \bar{\tau}) \cdot (s_1, \alpha) \cdot q \wedge I(s_1, \alpha, \tau)\}$$
- *cancel*: removing an anti-event/event pair from the window.  

$$\text{cancel}((s, b, h, s_c, np)) = \{(s, b, p \cdot q, s_c, np) \mid h = p \cdot (s_1, \bar{\tau}) \cdot (s_2, \tau) \cdot q\}$$

We now describe a set of properties to be checked on the transducer  $M'$ . Our implementation of this technique generates automatically CTL and LTL formulas that express these properties. For clarity reasons, the properties are described verbally here, omitting most of the actual formulas generated by the tool.

1. From every state in which the error flag is false, a state can be reached in which the window is not full:  $AG(\neg \text{error} \rightarrow EF(\neg \text{error} \wedge |h| < L))$
2. In every computation where the error flag remains false, every anti-transition in the history can be swapped with any “normal” transition following it in the window (so it can reach the end of the history queue).
3. (For every  $\tau$  that can be predicted): If  $\bar{\tau}$  is at the end of the window then  $\tau$  is enabled in the current state of  $M$ .
4. (For every  $\tau$  that can be predicted): If  $\bar{\tau}$  is not followed by  $\tau$  somewhere behind it in the window, then  $\tau$  will eventually be taken.
5. (For every  $\tau$ ): If the computation does not remain forever quiescent, then if  $\tau$  is in the window, it will eventually be chopped or will be cancelled with  $\bar{\tau}$ .
6. From every quiescent state with the error flag false, it is possible to remain quiescent (while reordering and chopping the window) until the window is emptied.
7. Along every path it is infinitely often possible to stop executing the system’s transitions (and perform history-related operations like predictions, swaps, chops), until the history is emptied or contains only anti-transitions:  $AG AF(\neg \text{error} \rightarrow E(\text{no-}\tau\text{-taken} \cup \text{no-}\tau\text{-in-}h))$ .
8. In every state  $s \in \Sigma$ , and for every  $\tau_1, \tau_2 \in \mathcal{T}$  such that  $I(s, \tau_1, \tau_2)$ , we have (i)  $\text{CondIndep}(s, \tau_1, \tau_2)$ , and (ii) if a third transition  $\tau_j \in \mathcal{J}$  is enabled in  $s$  and remains enabled after executing  $\tau_1$  and then  $\tau_2$ , then it is also enabled after executing  $\tau_2$  from  $s$ :  $en(s, \tau_j) \wedge en(\tau_1(s), \tau_j) \wedge en(\tau_2(\tau_1(s)), \tau_j) \rightarrow en(\tau_2(s), \tau_j)$ .

If branching-time property 1 holds, then the transducer can “simulate” every computation  $g$  of  $M$ , while the concatenation of the events chopped from the window forms a convenient run  $c$  such that  $c \sqsubseteq g$ . Linear-time properties 2,3 and 4 are needed to justify the predictions. For the chopped sequence of events to be equivalent to the original computation of  $M$ , they must have the same events. Properties 5 and 6 are needed to rule out cases when an event remains in the window forever. Branching-time property 6 deals with computations that become and remain quiescent. Branching-time property 7 is used to extend the reduction ( $c \sqsubseteq g$ ) into full equivalence ( $c \approx g$ ). Property 8 is an invariant that can also be checked on the original system  $M$  (without a transducer). It checks the requirement that the user-defined independence relation implies conditional independence, and a restriction needed for the equivalence to preserve justice.

**Theorem 1** *When verified on the transducer  $M'$ , properties 1-8 imply that every computation of  $M$  is equivalent to some convenient one.*

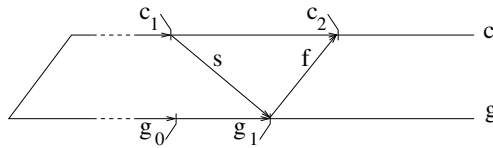
The full proof can be found in [1]. Here we present only a detailed outline of the proof. First, we define the main concept used to link the definition of the transducer with the equivalence we need to prove.

**Definition:** Given an infinite computation  $g$ , and an infinite sequence  $R = \{r_i\}_{i=0}^{\infty}$  of infinite computations, we say that  $R$  is a *reduction sequence* from  $g$  iff  $g = r_0$  and  $\forall i \geq 0 : r_i \equiv_{sw} r_{i+1}$ .

We say that a computation  $g'$  of  $M'$  follows a computation  $g$  of  $M$  if  $g$  is the projection of  $g'$  to the transitions and variables of  $M$ , and the *error* flag remains false along  $g'$ . Let us consider a computation  $g'$  that follows  $g$ . At any given time  $i$  along the execution of  $g'$ , a computation  $r_i$  can be built as the concatenation of the portion already chopped, with the contents of the window, followed by the part of  $g$  that did not execute yet (e.g., as depicted in Figure 2). If we do not consider predictions, it is easy to see that the sequence  $\{r_i\}_{i=0}^{\infty}$  is a reduction sequence:  $r_0 = g$ , and the only change from some  $r_i$  to  $r_{i+1}$  can be a swap of consecutive independent events. In general, if a reduction sequence from  $g$  converges, and its limit is  $c$ , then it can be proved that  $c \sqsubseteq g$ .

Property 1 implies that for every computation  $g$  there is a  $g'$  that follows it, so for every  $g$  there is a convenient run  $c$  such that  $c \sqsubseteq g$ , which is one half of what we need to prove. The predictions, as we saw, are just another way to prove swap equivalence, by moving anti-events forward in time instead of real events backwards. The net effect of a predict-swap-cancel sequence is a simple swap equivalence, so a reduction sequence from  $g$  can still be built from a computation  $g'$  that follows  $g$ . Of course, this requires properties 2, 3, and 4 to make sure the predictions are correct.

To prove full equivalence ( $c \approx g$ ), we need to prove  $g \sqsubseteq c$ , which means that infinitely often along  $g$ , there is a continuation that is swap-equivalent to  $c$ . We now explain why this is precisely what Property 7 implies. Consider point  $g_0$  in Figure 4. By Property 7,



**Fig. 4.** Proving full equivalence.

there is a later point  $g_1$  from which the transducer can stop executing transitions until (i) the window is empty, or (ii) it contains only anti-transitions. If the window is empty, then  $g$  and  $c$  have actually converged. If it contains only anti-transitions, this means that all the pending scenarios have been completed by predictions, and chopped out of the window. In the figure, the transitions in the segment marked with  $s$  are the ones starting new scenarios, those in the segment marked with  $f$  are the ones finishing pending scenarios. The contents of the window are the anti-transitions of those in the  $f$  segment. The actual occurrences of the events from  $f$  along  $g$  will appear after  $g_1$ , and cancel with their

respective predictions. Thus, a segment like  $f$  (connecting  $g$  to  $c$ ) *exists*, for every point  $g_0$  arbitrarily chosen along  $g$ , and this is what we need to prove the full equivalence.

The limit of a reduction sequence may not be a fair run. If we proved that the limit is an equivalent run, then the set of transitions that are taken infinitely often is the same in  $c$  as in  $g$ . The limit will be non-fair if some just transition which is never taken becomes enabled forever. This is avoided by the additional constraints on the user-defined independence relation introduced by Property 8. It is then easy to show by induction on the chain of swaps, that if  $\tau_j$  is enabled from some point onwards, then the same happens in any swap equivalent run. This can be used in turn to show that if  $\tau_j$  is enabled in  $c$  from some point onwards, then it must also be so in  $g$ . The proof is based on the fact that infinitely many paths connect  $c$  and  $g$  (in both directions), and converging paths are swap-equivalent.

## 4 Prototype Implementation - CNV

CNV (for CoNvenient), a prototype implementation of the described techniques, is based on a simple language for the description of fair transition systems, their convenient computations, the independence relation  $I$ , and additional configuration information such as the window size  $L$ , and predictions. The program `cnv2smv` translates a `.cnv` file into a description of the transducer and its proof obligations in the language of Cadence SMV, in which the verification and analysis of counterexamples is done. CNV includes optimizations meant to make the window smaller and use it efficiently:

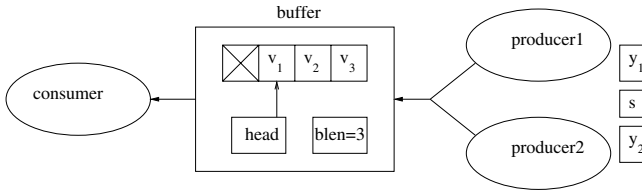
- (i) If the definition of the scenarios and of  $I$  depends only on an abstraction of the state information (e.g., on part of the state variables), and it is possible to update that abstract state with the effect of every transition, then it suffices to keep only the abstract state information in the window. An example is a queue, where the independence of *sends* and *receives* is affected only by the length of the queue, and not by its contents. A common example is a system where  $I(s, \tau_1, \tau_2)$  does not depend on  $s$ . It is then possible to have only transitions in the window, without any state information. CNV's input language supports the manual definition of an abstract state to be stored in the window.
- (ii) When a predicted transition completes a scenario so it can be chopped, instead of inserting the prediction and then chopping the whole scenario, the partial scenario is automatically replaced by the corresponding anti-transition. Similarly, if a transition is taken when its anti-transition appears at the end of the window, then instead of enqueueing the transition and then cancelling it with the anti-transition, CNV immediately removes the anti-transition from the window.
- (iii) To keep the window as small as possible, the transitions of the transducer generated by CNV are prioritized. Enabled transitions are executed in the following descending order of priority: chops, cancelling of predictions, swapping anti-transitions towards the end of the history, making new predictions, and finally (with the same priority) making swaps and executing transitions from  $M$ .

Apart from proving that every computation is equivalent to a convenient one (by answering “true”), CNV can in some cases provide a detailed reduction sequence from a specific computation  $g$ . If it is possible to describe a *single* computation  $g$  by an LTL formula, then CNV can be asked to check that there is *no* reduction from  $g$  to a convenient

computation. If this is not the case, the counterexample generated shows the reduction steps in detail.

The language of CNV allows the definition of scalar variables and a finite set of transitions, where each transition may be marked as just. For every transition, an enabling predicate and an assignment to every affected variable are defined. It is possible to define abstract state variables to be kept in the history, and to specify how they depend on the system state variables. The choppers currently supported by CNV are defined by giving a prioritized list of “convenient prefixes”(finite sequences of transition names) representing scenarios. The predictions are also given in a prioritized list. Each entry actually describes a sequence of predictions. For example:  $\text{PRED } Y \ Z \ \text{AFTER } W \ X$  will trigger the prediction of  $Y$  when the history has a prefix  $W \ X$ , and will then predict  $Z$  after the prefix  $W \ X \ Y$ , chopping the whole scenario immediately. The result of these two predictions on  $W \ X \ Q$  is  $\overline{Z} \ \overline{Y} \ Q$ .

**Application Example:** We use a schematic example to describe various parts of a methodology for which our technique can be useful. We consider a setting as the one shown in Fig.5, where two *producers* create complex values (maybe from an unbounded domain) and send them through a FIFO buffer, to a single *consumer*. The insertion of the value into the queue is not atomic, therefore the producers run Peterson’s mutual exclusion protocol [16] to avoid enqueueing at the same time. The convenient sequences are naturally defined as those in which every produced value is immediately enqueued, dequeued and consumed. The relation  $I$  we will define does not depend on the actual values being produced and transmitted. Therefore, equivalence based on it can be model checked by CNV, by ignoring those values altogether.



**Fig. 5.** Two mutually exclusive producers and one consumer with a FIFO buffer in between.

A valuable feature of CNV is its capability to verify Property 8 by generating and model checking an invariant of the original system (without any transducer). To prove commutativity, we cannot completely ignore the fact that different processes may overwrite each other’s values in the buffer. However, when proving the equivalence itself with the transducer, the actual values produced and sent through the buffer are not relevant. Therefore, we can remove the produced data bits from the transducer we use to check the equivalence itself (Properties 1-7).

The convenient computations of the system in Figure 6, as described before, should be those in which transition  $l_2$  is immediately followed by  $l_3, l_4, c_1, c_2$  (and similarly for  $m_2$ ). Proving the reduction in a single step means that the history queue we may need for the proof could be quite long, since there must be enough room in the history

```

Initially: head=blen=y1=y2=0, s=1
Producer 1: | | Producer 2: | | Consumer:
loop forever do | | loop forever do | | loop forever do
  l1:<PRODUCE v1;y1:=1;s:=1> | | m1:<PRODUCE v2;y2:=1;s:=2> | | c1:await blen>0
  l2: await (y2=0\s!=1) /\ | | m2: await (y1=0\s!=2) /\ | | c2:<CONSUME
b[head];
      (blen<N) | | (blen<N) | | head:=(head+1) mod
N;
  l3: STORE(v1,b[head+blen]) | | m3: STORE(v2,b[head+blen]) | | blen:=blen-1>
  l4: <blen:=blen+1;y1:=0> | | m4: <blen:=blen+1;y2:=0> | | end loop
end loop | | end loop | |

```

**Fig. 6.** Pseudo code of the system

for the convenient prefixes and for any anti-transitions that may be generated before a convenient prefix is created. The computational complexity of the verification depends heavily on the length of the history, so we prefer to perform the reduction in stages.

In a first stage, we prove that every computation is equivalent to one in which the sequences  $(l_2, l_3, l_4)$ ,  $(m_2, m_3, m_4)$ , and  $(c_1, c_2)$  are executed atomically. This basically proves that there is mutual exclusion between the two producers in their access to the buffer. Our independence relation  $I$  depends on the state (e.g.,  $l_2$  and  $c_2$  are independent if the buffer is not full). According to  $I$ , we chose to store in every history element the variables  $y_1, y_2, s$  and  $blen$  (the buffer's current length). The chopper recognizes the following prefixes as convenient:  $(l_1), (l_2, l_3, l_4), (m_1), (m_2, m_3, m_4), (c_1, c_2)$ . The predictions we used are: "PRED  $l_3 l_4$  AFTER  $l_2$ ", "PRED  $m_3 m_4$  AFTER  $m_2$ ", and "PRED  $c_2$  AFTER  $c_1$ ". A window of length 4 was sufficient to prove the reduction, for a buffer length  $N=2$ . Verifying the equivalence (Properties 1-7) on the data-less version of the transducer took 105 minutes and 303MB of memory<sup>4</sup> on a 1.2GHz machine, and verifying the independence relation (Property 8) on the original system (i.e., with the buffer's data bits but without the transducer) took less than 10 seconds.

In the second stage, we model these sequences as atomic transitions (and we adjust  $I$  accordingly), and prove that every computation is equivalent to one in which every occurrence of  $l_2$  (which now includes  $l_3$  and  $l_4$ ) is immediately followed by  $c_1$  (which now includes  $c_2$ ), and similarly for  $m_2$ . This time, a window of length 3 was sufficient to prove the reduction, for the same buffer length. Verifying the equivalence on the data-less version took 2 minutes and 14MB of memory, and verifying the independence relation on the full version of the original system without the transducer took less than 5 seconds.

To further test this approach, we split the first stage into two sub-stages: In the first one we prove equivalence to computations in which the transition pairs  $(l_3, l_4)$ ,  $(m_3, m_4)$  and  $(c_1, c_2)$  occur atomically. The second sub-stage proves equivalence to computations in which  $(l_2, l_3, l_4)$  and  $(m_2, m_3, m_4)$  occur atomically, as in the first big stage above. The first sub-stage took 36 minutes and used 89MB and the second one took 6 minutes and used 37MB, so together they took 42 minutes instead of the 105 minutes required previously, and used less than one third of the memory space.

This example's sources, the `cnv2smv` tool and a skeleton input file appear in [1].

<sup>4</sup> This is attributed to the asynchronous nature of the system, which is not particularly suited to the SMV model checker. An additional "running" variable is created by SMV for every transition of the transducer, and the resulting BDDs are not as compact as for hardware verification.

## 5 Discussion and Future Work

This paper has presented, for the first time, a general framework and tool for model checking conformance of a model to scenario-based or transaction-based specifications. The automatically generated transducer, including the window into the history and the added transitions for chopping and swapping events in the window, of course adds to the size of the state-space and transition relation. This should be viewed as the price for making the strong claim of computation equivalence in such specifications, analogous to the cross-product of the model with an automaton seen in automata-based model checking of linear-time temporal properties.

Although in theory the number of state variables is increased by  $L \times (|V| + \log |\mathcal{T}|)$ , the effective increase can be greatly reduced by exploiting optimizations, just as in other model checking tasks. Even in the prototype implementation described here, the transducer's transitions are prioritized in order to minimize the length of the needed window, and facilities are provided for abstracting away unneeded state variables from the history, reducing the amount of information kept there to the minimum needed to support the history-related operations. Usually, this means that each window element has only a few state bits in addition to the  $\log |\mathcal{T}|$  needed to represent a transition. The possibility of using predictions to chop a scenario before it has completed, and using an anti-event to ensure that predicted events eventually occur, is also crucial in reducing the size of the window.

Moreover, it should be noted that not all of the transducer states exploit the full size of the window, and many transitions simply cannot occur from some states. A careful tuning of the variable ordering could use this fact to facilitate the construction of more compact BDDs. An explicit-state model checker, in principle, could also exploit this fact by using a dynamically sized representation of the states reached during the explicit search.

In future work, the influence on complexity of modelling decisions, such as how to abstract the state, and which predictions to make, needs to be better understood. Further compressing the contents of the window is crucial. Splitting the description of a long scenario into shorter sub-scenarios, and proving the equivalence in several steps, as seen in the example, also offers significant benefits. Other research directions include further development of notations for expressing scenarios, including connections to existing modelling techniques like UML.

## References

1. <http://www.cs.technion.ac.il/Labs/ssdl/pub/CNV>.
2. R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 340–351. Springer-Verlag, 1997.
3. T. Braun, A. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving sequential consistency by model checking. In *Proc. 6th IEEE High Level Design Validation and Test Workshop, (HLDVT'01)*, pages 103–108, December 2001.
4. M. Glusman and S. Katz. A mechanized proof environment for the convenient computations proof method. *Formal Methods in System Design*. To appear. Available at [http://www.cs.technion.ac.il/Labs/ssdl/pub/conv\\_PVS](http://www.cs.technion.ac.il/Labs/ssdl/pub/conv_PVS).

5. M. Glusman and S. Katz. Mechanizing proofs of computation equivalence. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, (CAV'99), volume 1633 of *LNCS*, pages 354–367. Springer-Verlag, 1999.
6. T. A. Henzinger, S. Qadeer, and S. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, (CAV'99), volume 1633 of *LNCS*, pages 301–315. Springer-Verlag, 1999.
7. G. J. Holzmann and D. Peled. The state of SPIN. In *Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 385–389. Springer-Verlag, 1996.
8. B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Distributed Computing*, 12:129–149, 1999.
9. S. Katz. Refinement with global equivalence proofs in temporal logic. In D. Peled, V. Pratt, and G. Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–78. American Mathematical Society, 1997.
10. S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.
11. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
12. M. Kwiatkowska. *Fairness for Non-Interleaving Concurrency*. PhD thesis, Dept. of Computing Studies, Leicester, 1989.
13. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Safety*. Springer-Verlag, 1995.
14. A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and editors G. Rozenburg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *LNCS*, pages 279–324. Springer-Verlag, 1986.
15. Ken L. McMillan. *Getting Started With SMV*. Cadence Berkley Labs, 2001 Addison St. Berkley, CA, March 1999.
16. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
17. A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd. Workshop on Computer-Aided Verification*, volume 531 of *LNCS*, pages 156–165. Springer-Verlag, 1990.
18. P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *LNCS*, 1993.



# Deductive Verification of Advanced Out-of-Order Microprocessors<sup>\*</sup>

Shuvendu K. Lahiri and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, PA  
shuvendu@ece.cmu.edu, Randy.Bryant@cs.cmu.edu

**Abstract.** This paper demonstrates the modeling and deductive verification of out-of-order microprocessors of varying complexities using a logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU). The microprocessors support combinations of out-of-order instruction execution, superscalar operation, branch prediction, execute and memory exceptions, and load-store buffering. We illustrate that the logic is expressive enough to model components found in modern processors. The paper describes the challenges in modeling and verification with the addition of different design features. The paper demonstrates the effective use of automatic decision procedure to reduce the amount of manual guidance required in discharging most proof obligations in the verification. Unlike previous methods, the verification scales well for superscalar processors with wide dispatch and retirement widths.

## 1 Introduction

In the last few years, several different techniques have been employed for the formal verification of advanced microprocessors. These include the use of symbolic model checking [3], compositional model checking [10], deductive verification methods based on theorem proving [1,9,13] and symbolic simulation with decision procedures for quantifier-free first order logic [5,7].

Most of the previous efforts in verifying microprocessors with unbounded resources (e.g., with an arbitrarily large reorder buffer or load-store queue) are based on deductive verification methods and involve a general purpose theorem prover (PVS [12], ACL2 [4]) to discharge the proof obligations in the verification. This involves writing down large proof scripts to systematically prove the invariants. This is both time-consuming and requires very careful understanding of the theorem provers. Moreover, the lack of counter-examples for failed proofs renders the invariant strengthening method difficult and relies on the ingenuity of the user.

In earlier work [11], we used the logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions (CLU), to model and verify a simple

---

<sup>\*</sup> This research was supported in part by the Semiconductor Research Corporation, Contract RID 1029.001.

out-of-order execution unit. Deductive verification was used to prove the correctness of the processor by establishing a set of *refinement maps* [1,10], to show that the implementation refines the specification. The refinement maps specify the correctness of signals or values in the implementation with respect to a sequential Instruction Set Architecture (ISA) model. All the proof obligations were discharged automatically using sound quantifier instantiation techniques and the decision procedure for CLU. Further, the use of the restricted logic enabled us to produce counterexamples for the failed proofs.

In this work, we investigate if the logic of CLU is expressive enough to model and verify out-of-order processors with advanced features such as speculation, superscalar behavior and buffered memory instructions. We show how the addition of new instruction types and design features add to the modeling and verification challenges using CLU. Unlike prior work in the verification of processors with unbounded resources, we demonstrate the verification of superscalar processors, where multiple instructions can be dispatched and retired simultaneously.

Category	Unbounded Resources	Speculation, Exceptions	Data Memory	Methodology
Sawada and Hunt. [13]		×	×	Deductive verification with ACL2
Skakkebaek et al. [14]	×			Correspondence checking with manual abstraction
Berezin et al. [3]	×			Finite State Model Checking
Arons et al. [1]	×	×		Deductive Verification with PVS
Hosabettu et al. [9]	×	×	×	Deductive Verification with PVS
Jhala, McMillan [10]	×	×	×	Compositional Model Checking
Velev [15]				Correspondence Checking
Lahiri et al. [11]	×			Deductive Verification with UCLID
Current	×	×	×	Deductive Verification with UCLID

**Fig. 1. Previous efforts for out-of-order processor verification.**

**Related Work.** Figure 1 shows a chronological listing of different approaches to out-of-order processor verification along with the features of the processors verified. In the next few paragraphs, we concentrate on previous works that verify an out-of-order processor supporting speculation, exceptions and memory instructions with load-store queues.

Jhala and McMillan [10] use compositional model checking (with Cadence SMV) to verify refinement maps between an out-of-order processor and a sequential ISA model. A finite state abstraction is generated by exploiting temporal case-splitting, data-type reduction and symmetry. Model checking is then used on the abstract state space to verify the properties. Although more automatic than the deductive verification based approaches, the method still requires the user to explicitly decompose the proof into smaller lemmas to alleviate the state explosion. Besides, the approach relies heavily on symmetry in the system. It can be ineffective for many practical systems, where symmetry is broken by the presence of priority encoders (e.g. processors with wide dispatch and retire widths) or the heterogeneity of deep pipelines. Since our verification does not

explicitly use symmetry, our approach is robust in the presence of asymmetry in the design as we demonstrate in the verification of the superscalar processors.

Deductive verification based methods [13,9,1] use general purpose theorem provers to establish the correctness of microprocessors. Sawada and Hunt [13] use the ACL2 theorem prover to verify the correctness of microprocessors with bounded retirement and load-store buffers. They limit at most 15 instructions in the pipeline at any time. They use a trace-table based intermediate representation called MAETT to record redundant information for committed and in-flight instructions. This intermediate abstraction is used to specify invariants to relate the ISA and the pipelined implementation. It should be mentioned that they model external interrupts, which no one else (including our current models) handle at present. On the other hand, our work considers unbounded resources and requires significantly fewer lemmas compared to the almost 4000 lemmas in Sawada and Hunt’s case. Hosabettu et al. [9] use a *completion function* approach to complete all the partially executed instructions in the system. The “flushed” state of the implementation is then compared against the ISA model. The method requires the user to construct an inductive completion function for the different instruction types (in different stages of execution) and then compose the different completion functions to obtain the abstraction function. The PVS [12] theorem prover is used to discharge the proofs. Both Sawada et al. and Hosabettu et al. use a variant of Burch-Dill method of “flushing” the pipeline and prove the equivalence of an empty pipeline with the ISA state. Our approach differs from these methods in two ways. First, we use refinement maps (similar to Arons et al. [1]) between the implementation and the sequential ISA to prove the correctness of the processor. Second, the use of decision procedure reduces the burden of proving most of the proof obligations that arise during the verification.

The rest of the paper is organized as follows. In Section 2, we provide a brief overview of the tool UCLID. The section outlines the logic and the method of verification. In Section 3, the description and modeling of the different out-of-order processors are presented. Finally, in Section 4, the verification is described. This includes the definition of different auxiliary fields, the refinement maps, description of invariants and their proofs.

## 2 Background

The tool UCLID [6,11] uses the logic of CLU (described in Fig 2) to model and verify systems with unbounded resources. CLU is a fragment of quantifier-free first order logic extended with increment (**succ**), decrement (**pred**), equality and inequality operations over *terms* (integer expressions). *ITE* denotes the “if-then-else” constructor to choose between two terms depending on a boolean control. The uninterpreted function and predicate symbols can be used to specify an arbitrary value for the function state variables in the most general state or to abstract out combinational blocks [5,10] (e.g., the ALU) in the system.

$$\begin{aligned}
\text{bool-expr} &::= \mathbf{true} \mid \mathbf{false} \mid \neg \text{bool-expr} \mid (\text{bool-expr} \wedge \text{bool-expr}) \\
&\quad \mid (\text{int-expr} = \text{int-expr}) \mid (\text{int-expr} < \text{int-expr}) \\
&\quad \mid \text{predicate-expr}(\text{int-expr}, \dots, \text{int-expr}) \\
\text{int-expr} &::= \text{int-var} \mid \text{ITE}(\text{bool-expr}, \text{int-expr}, \text{int-expr}) \\
&\quad \mid \mathbf{succ}(\text{int-expr}) \mid \mathbf{pred}(\text{int-expr}) \\
&\quad \mid \text{function-expr}(\text{int-expr}, \dots, \text{int-expr}) \\
\text{predicate-expr} &::= \text{predicate-symbol} \mid \lambda \text{int-var}, \dots, \text{int-var}. \text{bool-expr} \\
\text{function-expr} &::= \text{function-symbol} \mid \lambda \text{int-var}, \dots, \text{int-var}. \text{int-expr}
\end{aligned}$$

**Fig. 2. CLU Syntax.** Expressions can denote computations of Boolean values, integer values, or functions over integers yielding Boolean or an integer value.

The presence of lambda expressions not only subsumes the interpreted **read** and **write** operators for unbounded arrays [6] but also allows us to model *parallel-update* memories. In a parallel-update memory  $M$ , an arbitrary number of entries satisfying some predicate (say  $P$ ) can get updated (with some function  $D$ ) in one step as follows:

$$M' = \lambda i. \text{ITE}(P(i), D(i), M(i))$$

Here  $M'$  denotes the next state of the memory  $M$ . This is very important for modeling the forwarding of result data to an arbitrary number of dependent instructions in an out-of-order processor [11]. The **succ** and **pred** operations allow us to model ordered data structures such as queues of arbitrary length, by performing appropriate increment or decrement operations for the head and tail pointers for the queue. Various other data structures including content-addressable memories, and circular queues can be expressed in CLU [11].

A well-formed CLU formula<sup>1</sup>  $F$  is *valid* (denoted as  $\models F$ ) when it is true under all possible interpretations of symbols in  $F$ . The *decision procedure* for CLU checks the validity of a well-formed formula  $F$  by a validity-preserving translation to a propositional formula and using Boolean techniques to evaluate the formula. Counterexamples for invalid formulas are mapped to the state-variables to produce counter-example traces. Details about the logic and decision procedure can be found in earlier work [6].

**Deductive Verification.** A system is verified by proving a set of invariants inductively. Let  $\mathcal{I}_1(s), \dots, \mathcal{I}_n(s)$  be a set of invariants on a state  $s$  of the system. To prove the base case, we show:

$$\models \mathcal{I}_1(s_0) \wedge \mathcal{I}_2(s_0) \dots \wedge \mathcal{I}_n(s_0) \quad (1)$$

<sup>1</sup> An integer variable  $x$  is said to be *bound* in expression  $E$  when it occurs inside a lambda expression for which  $x$  is one of the argument variables. An expression is *well-formed* when it contains no unbound variables.

where  $s_0$  is the start (reset) state of the system. For the induction step, we start with a general state  $s$  and then symbolically simulate the system for one step to obtain state  $\delta(s)$ , where  $\delta$  is the transition function. We then show:

$$\models \mathcal{I}_1(s) \wedge \dots \wedge \mathcal{I}_n(s) \implies \mathcal{I}_i(\delta(s)) \quad (2)$$

for each  $1 \leq i \leq n$ . Usually, the property to be verified is specified as one of these invariants. The other invariants are added (manually) to *strengthen* the inductive invariant.

Similar to our previous work [11], we restrict the invariants to be of the form  $\forall x_1 \dots \forall x_k. \Phi(x_1, \dots, x_k)$ , where  $x_1, \dots, x_k$  are integer variables *free* in the CLU formula  $\Phi(x_1, \dots, x_k)$ . To prove that such an invariant is inductive (in Equation 2), we need to decide formulas of the form

$$\models \forall x_1 \dots \forall x_m \Psi(x_1, \dots, x_m) \implies \forall y_1 \dots \forall y_k \Phi(y_1, \dots, y_k) \quad (3)$$

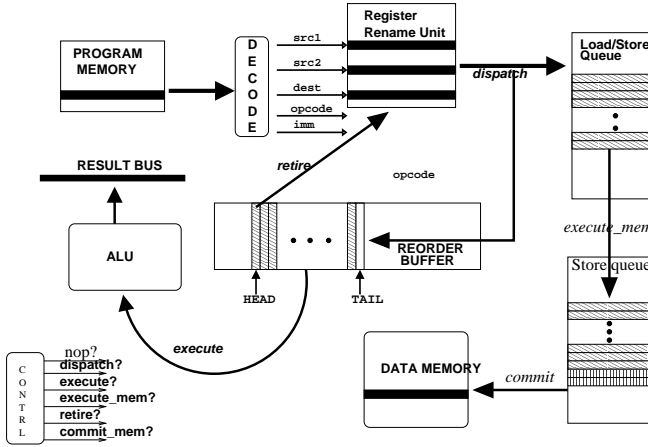
Since checking validity for first-order formulas of the form (3) is undecidable [8], we perform a sound translation of the formula in (3) to a CLU formula, which can be checked by a decision procedure.

The reduction of the formula in Equation 3 involves replacing the universal quantifiers to the right of the implication with fresh *skolem constants*  $\hat{y}_1, \dots, \hat{y}_k$ . The antecedent of the implication,  $\Psi$  is instantiated over terms appearing in the consequent  $\Phi$ . Details of the quantifier instantiation can be found in earlier work [11]. If  $\mathcal{A}_{x_i}$  denote the set of terms to instantiate the variable  $x_i$ , then the final formula becomes:

$$\left[ \bigwedge_{t_1 \dots t_m \in \mathcal{A}_{x_1} \times \dots \times \mathcal{A}_{x_m}} \Psi(t_1, \dots, t_m) \right] \implies \Phi(\hat{y}_1, \dots, \hat{y}_k) \quad (4)$$

### 3 OOO Processor Description

OOO (shown in Figure 3) is a model of an out-of-order processor that employs Tomasulo's algorithm, supports speculative execution, exceptions, memory instructions and load-store queues. On each step, the system arbitrarily chooses between six different operations: *dispatch*, *execute*, *execute\_mem*, *retire*, *commit\_mem* and *nop*. During *dispatch*, an instruction is dispatched to the Reorder Buffer (ROB). An entry is also created in the Load-Store Queue (LSQ) if it is a memory instruction. During *execute*, a ready arithmetic or branch instruction in the ROB (with all operands valid) is scheduled for execution. During *execute\_mem*, the memory instruction at the head of the LSQ is executed. Load instructions can obtain their data from the preceding store instructions when there is an address match in the Store-Queue (STQ). Otherwise the data comes from data memory. The store instructions are enqueued into the STQ after execution with the store address and the data. During *retire*, the instruction at the head of the ROB is retired and the register state is updated. For store instructions, the entry in STQ is marked as *retired*. The actual memory update takes place during a *commit\_mem* operation.



**Fig. 3. An Out-of-order processor with exceptions, speculation and memory instructions.**

Instructions are retired in program order to track precise exceptions. An exceptions can be raised as a result of either arithmetic instruction execution, an illegal data address for a memory instruction or an illegal address for a branch instruction. If the currently retiring instruction raises an exception, the ROB and LSQ are flushed. All of the non-retired entries in the STQ are also dropped and all of the registers in the register file set the `reg.valid` bit to **true**.

The main components of the design are (i) a Reorder Buffer for inorder completion and precise exceptions, (ii) a Register Rename unit for out of order execution, (iii) a Load-store queue for non-executed memory instructions, (iv) a Store-Queue for stores which have finished execution. Below we describe the modeling of each of them in some detail.

**Reorder Buffer.** The reorder buffer (ROB) is modeled as a queue with head and tail pointers. It also supports simultaneous update of a subset of entries if an operand tag matches the currently executing instruction. Each entry in the ROB contains fields for the instruction type `rob.itype`, opcode `rob.opcode`, destination register `rob.dest`, immediate value `rob.imm`, operand values `rob.src1val`, and `rob.src2val`, program counter `rob.pc`, prediction flag `rob.mispredict` and target address `rob.target`. To indicate if the operands are ready, it maintains `rob.src1valid` and `rob.src2valid` bits and fields `rob.src1tag` and `rob.src2tag` indicating the instructions producing the data. The `rob.valid` bit indicates the instruction has finished execution. The field `rob.value` stores the write-back value for arithmetic and load instructions.

**Register Rename Unit.** The register rename unit consists of an infinite array of  $\langle \text{reg.valid}, \text{reg.val}, \text{reg.tag} \rangle$  tuples. Every time an arithmetic or load instruction is dispatched with destination register  $d$ , the `reg.valid` bit for  $d$  is set to **false** and the tag for the dispatched instruction is recorded in the `reg.tag` field. A retiring instruction sets the `reg.valid` bit to **true** if its tag matches the tag stored in the register. A retiring mispredicted branch or an

instruction with an exception, causes all of the registers to simultaneously reset their `reg.valid` bits to `true`.

**Load-Store Queue.** The load-store queue (LSQ) maintains the non-executed memory instructions in program order. Memory instructions are enqueued in the LSQ during *dispatch* and dequeued during *execute\_mem*. Each LSQ entry contains a pointer to the corresponding ROB entry, called `lsq_rob_ptr`. The result of execution for a load instruction is written back in the `rob.value` field of the ROB entry pointed to by the `lsq_rob_ptr`. For store instructions, the data in `src2val` of the corresponding ROB entry and the address is enqueued into the STQ.

**Store Queue.** The store queue (STQ) maintains the store instructions which have finished execution, in program order. A pointer `stq_retire_ptr` points to the *first* non-retired instruction in the STQ. During a squash operation (due to an exception or misspeculation for the retiring instruction), all of the non-retired instructions in the STQ are abandoned.

Since the system supports forwarding data from a preceding store instruction to a dependent load, there is a need to (i) find if an address is present in the STQ and (ii) identify the latest entry with the matching address. Since an associative lookup can't be modeled directly for infinite structures in UCLID, we maintain a map structure, `stq_pos`, which maps each memory address to the position of the *latest* entry (if present) in the STQ. An address  $A$  is present in the STQ iff  $(stq\_head \leq stq\_pos(A) < stq\_tail) \wedge (stq\_addr(stq\_pos(A)) = A)$ .

This way of modeling the associative lookup however has one problem. During a squash operation, the non-retired instructions in the STQ are abandoned. This would require resetting `stq_pos` for a given address  $A$  to the *latest* retired entry in the STQ which matches the address  $A$  (if present). To circumvent this problem, we maintain another map `stq_nonspec_pos` which maps an address  $A$  to the latest retired entry in the STQ, which matches  $A$  (if present). It is updated every time a store instruction is retired. During squash, the map `stq_nonspec_pos` is copied onto `stq_pos`. Note that this map is not required if stores are committed to memory during retire.

## 4 Verification of the Processors

We establish refinement maps to prove the correctness of the implementations with respect to a sequential instruction set architecture (ISA). The ISA state components consists of the register file, data memory and, program counter.

We built the models of the out-of-order processors incrementally to study the effect of different features on both the modeling and the verification effort. Figure 4 shows the different models and the features that were *added* with each model.

First, we describe the various auxiliary data structures and variables added and later express the correctness criteria for the augmented system.

## 4.1 Auxiliary Fields

We had to add a number of auxiliary variables to the system to enable the verification. These variables do not affect the system operation, but are added for two principal reasons, discussed below.

Model	Features
OOO.base	Out-of-order execution, inorder retirement
OOO.ex	Arithmetic exceptions
OOO.ex_br	Branch Prediction
OOO.ex_br_mem_simp	Memory instructions, LSQ, STQ, Stores commit during <i>retire</i>
OOO.ex_br_mem	Stores commit during <i>commit_mem</i>

Fig. 4. Description of different models.

First, we add auxiliary variables to express invariants about the correctness of values of variables in the system (similar to other works [1,10]). These additional state variables, called *shadow* variables in our case (prefixed with **shdw.**), predict the correct value for some actual state variables. For instance, the shadow entry **shdw.src1val** predicts the correct value for the state variable **rob.src1val** — for any index  $t$  in the ROB, if **rob.src1val**( $t$ ) contains a valid entry, then **rob.src1val**( $t$ ) = **shdw.src1val**( $t$ ). The *shadow* variables for an instruction are usually updated by the ISA machine during *dispatch* [11].

Second, we need to add additional variables to express an invariant of the form  $Q_1x_1, \dots, Q_nx_n. \Phi(x_1, \dots, x_n)$ , where  $\Phi(x_1, \dots, x_n)$  is a CLU formula and at least one of the  $Q_i$  is  $\exists$ . For instance, consider the invariant — *for every non-executed memory instruction  $t$  in the ROB, there exists a “corresponding” entry in the LSQ*. This is hard to express without existential quantifiers. We maintain a pointer **aux.rob\_lsq\_ptr** for each ROB entry to point to the corresponding entry in the LSQ. We can restate the invariant as — *for every non-executed memory instruction  $t$  in the ROB, **aux.rob\_lsq\_ptr**( $t$ ) is present in the LSQ*. The variables in this category are prefixed with **aux.**, e.g. (**aux.rob\_lsq\_ptr**). These variables essentially act as *witnesses* for the existential quantifiers.

Below we describe the auxiliary fields that were added with each design feature.

**1. OOO.base.** The auxiliary fields required were **shdw.src1val**, **shdw.src2val** and **shdw.value** for expressing the correct values of the data operands and result in each ROB entry.

**2. OOO.ex, OOO.ex\_br.** First we maintained a pointer into the ROB, named **shdw.exnmpred\_tag** to keep track of the earliest instruction that will raise an exception or cause a misprediction. We also required a map **shdw.reg\_tag**, which maps a register to the *latest nonspeculative* instruction in the ROB that modifies the register. Notice that in the presence of misprediction or exceptions, the instruction in the **reg\_tag** may not be the last instruction to write into the register before flushing the system.

**3. OOO.ex\_br\_mem\_simp.** The addition of load and store instructions required the addition of the largest number of auxiliary variables. First, with



each ROB entry, we maintain two additional pointers, `aux.rob_lsq_ptr` and `aux.rob_stq_ptr` to point to an entry in the LSQ and STQ respectively. Second, we also add *inverse* pointer `aux.stq_rob_ptr` from a STQ entry to the corresponding ROB entry. Note that `lsq_rob_ptr` is already present in the actual model. Third, a map `shdw.mem_tag` is added which maps each memory address to the *latest* non-speculative instruction in the ROB which modifies the address. As we shall see later, this is required to express the refinement map for the data memory. Fourth, for every load instruction in the ROB, we maintain two pointers `shdw.ld_tag` and `aux.ld_stq_ptr`. The first pointer points to the store instruction (if any) in the ROB that would forward the data (due to an address match) to the load. The second pointer points to the STQ entry that forwards the data to the load.

**4. OOO.ex\_br\_mem.** We did not require any further auxiliary structure to prove this model.

## 4.2 Correctness via Refinement Maps

The correctness of the implementation is proved by establishing three refinement maps with the ISA model.

The correctness for the register file is established by the following lemma:

$$\forall r : ISA.rf(r) = \begin{cases} reg.val(r) & \text{if } reg.valid(r) \text{ is } \mathbf{true} \\ - & \text{Otherwise} \end{cases}$$

The lemma states that if a register is not the destination of any of the instructions in the ROB, then the values in the implementation model and the ISA model are the same.

For the data memory, recall that `shdw.memtag` maps a memory address  $a$  to a position in the ROB (if present), containing the latest nonspeculative store instruction to write to address  $a$ . Thus, if there are no (non-retired) non-speculative instructions in the ROB which modifies  $a$  and there are no (retired) instructions in the STQ which modifies  $a$ , then both the ISA memory and the implementation memory should have the same value for  $a$ . The refinement map for data memory can be stated as:

$$\forall a : ISA.mem(a) = \begin{cases} mem(a) & \text{if } shdw.memtag(a) \text{ is not present in ROB and} \\ & a \text{ is not present in the STQ} \\ - & \text{Otherwise} \end{cases}$$

Similarly, there is a refinement map for the program counter:

$$ISA.pc = \begin{cases} pc & \text{If } shdw.exn_mpred\_tag \text{ is not present in ROB} \\ & \text{and } squash \text{ is } \mathbf{false} \\ exn\_pc & \text{If } squash \text{ is } \mathbf{true} \\ - & \text{Otherwise} \end{cases}$$

Note that `shdw.exn_mpred_tag` is present in the ROB iff some instruction in the ROB would raise an exception or cause misprediction. The signal `squash` is asserted after an exception is raised by the retiring instruction.

### 4.3 Invariants

In this section, we shall discuss the main categories into which we classified the invariants. The main categories of invariants are:

**1. Consistency Invariants.** These invariants express the relationship between the state variables of the actual system. These invariants can be stated without the help of auxiliary or shadow structures. For instance, the invariant that *every executed instruction has ready operands* can be stated without adding any auxiliary information to the model.

**2. Ordering Invariants.** Since the model contains three ordered data structures (ROB, LSQ, STQ), it is very important to maintain the program order of entries in the three queues. For example, if an instruction  $I_1$  precedes another instruction  $I_2$  in the LSQ, then  $I_1$  should precede  $I_2$  ROB as well.

**3. Bijective Invariants.** These invariants establish the relationship between two functions that act almost as *inverses* of each other. Examples include the pair of maps `aux.rob_stq_ptr` and `aux.stq_rob_ptr`. For any valid STQ index  $x$ , `aux.rob_stq_ptr(aux.stq_rob_ptr(x)) = x`. For any ROB index  $y$  with an executed store instruction, `aux.stq_rob_ptr(aux.rob_stq_ptr(y)) = y`.

**4. Value Invariants.** These invariants mainly specify the correctness of the data values in the model with reference to the *shadow* (predicted) values. For instance, for an executed arithmetic instruction, the value in `rob.value` should equal the `shdw.value` field for the same ROB entry. Value invariants involving memory instructions are more involved.

**5. PC Invariants.** These invariants specify the primary and auxiliary invariants for the correctness of the program counter. The invariants relate the program counter (`pc`) and the exception program counter (`exn_pc`) with the program counter of the ISA model in the presence and absence of misprediction or exception.

**6. Misprediction Invariants.** These invariants state the relationship of the `shdw.exn_mpred_tag` with the instructions in the ROB. For example, if any instruction in the ROB would raise an exception or be mispredicted, then the `shdw.exn_mpred_tag` points to the earliest such instruction in the program order.

**7. Register Tag Invariants.** These invariants relate the `reg.tag` with the shadow entry `shdw.reg.tag`. For example, if none of the instructions in the ROB raises an exception or is mispredicted, then `reg.tag(r) = shdw.reg.tag(r)`, for any register  $r$  that would be modified by an instruction in the ROB.

**8. Memory Tag Invariants.** These invariants relate the different maps for the memory instructions, namely `shdw.mem_tag`, `shdw.ld_tag`, `aux.ld_stq_ptr`, `stq_pos`, `stq_nonspec_pos` and `stq_retire_ptr`. These are the most involved invariants for the verification of the processor models with memory instructions.

**9. Other Invariants.** These invariants cannot be categorized into one of the classes mentioned above. They were mostly obtained from failed proofs after analyzing the counterexamples.

Figure 5 illustrates the classification of the invariants into the different logical categories described above for each of the benchmarks. The purpose of the classification is to understand the complexity of invariants for different parts of the design. The number of value invariants grows dramatically when we introduce memory instructions in the model, due to the need to define the correct values for load instructions in the presence of store forwarding. Moreover, since we need the additional map `stq_nonspec_pos` for model `000.ex_mem_br`, we have more value and memory-tag invariants for this model compared to the model `000.ex_mem_br_simp`.

Technique	000.base	000.ex	000.ex_br	000.ex_br_mem_simp	000.ex_br_mem
Consistency	5	5	7	6	6
Ordering	-	-	-	6	6
Bijective	-	-	-	4	4
Value	6	7	7	10	13
PC	3	4	7	8	8
Misprediction	-	8	8	8	8
Register Tag	-	8	8	8	8
Memory Tag	-	-	-	15	16
Other	3	2	2	2	2
Total #	17	34	39	67	71

Fig. 5. Classification of the number of invariants for different models.

#### 4.4 Proving the Invariants

The invariants are proved by the method described in Section 1. This method of instantiating the quantifiers with concrete sub-terms in the formula often generates the necessary terms to prove the valid formulas.

The number of combinations to instantiate can be exponential in the number of bound variables in the antecedent of Equation 3. The bound variables are indices to the different memories and unbounded arrays in the system. For the models without memory, the bound variables are register identifiers and ROB indices and the maximum number of combinations to instantiate was limited to 14 in these cases. With the introduction of LSQ and STQ, we needed up to 8 bound variables — to include LSQ and STQ indices and memory address. The number of combinations to instantiate increased up to 28800 for `000.ex_br_mem`. It is crucial to have a fast decision procedure for the large cases.

For most of the models, all the invariants were proved using the instantiation schemes described in Section 1. For the more complex models with multiple ordered structures, we had to resort to manual instantiation of a few invariants — 4 invariants for `000.ex_br_mem_simp` and 8 invariants for `000.ex_br_mem`. In most of these cases, at most two additional terms were needed. The terms were obtained easily by inspecting the counterexamples produced during the failed attempts.

Figure 6 summarizes the verification effort for the different models by several criteria. Note that the final models with the memory instructions are significantly more complicated to prove because of the large number of ordered data structures (LSQ and STQ in addition to ROB). Proving `000.ex_br_mem` is more complicated than `000.ex_br_mem_simp` because the presence of the retired instructions in the STQ<sup>2</sup>. Some of the criteria (# of invariants, person-effort) listed in Fig 6 are

Category	000. base	000. ex	000. ex_br	000. ex_br_mem_simp	000. ex_br_mem
# of invariants	17	34	39	67	71
Time Taken to Prove (sec)	54	235.76	403	1594.24	2200
UCLID Proof Script Size (KB)	9.91	20.06	23.59	68.67	66.79
Total Time Spent (Person Days)	2	5	2	15	10

**Fig. 6. Proof Effort for different models.** Time taken to prove denotes the time taken by UCLID to prove all the invariants inductive. Proof script size consist of the definition of invariants, auxiliary state variables and the proofs. The number of “Person Days” is the added effort for each model and is not cumulative.

very subjective and would differ from user to user depending on the knowledge of the design, dexterity with the tool or theorem prover and his/her ingenuity. But the ability to relieve the user from proving most of the proof obligations, results in a much smaller proof script size compared to previous attempts using a general-purpose theorem prover. For processors comparable to `000.ex_br_mem`, proof-script sizes for Sawada and Hunt [13] and Hosabettu et al. [9] are 2300KB and 1909KB respectively, as reported in [10].

One of the important contributions of this work is that we can use the automation and efficiency of the integer decision procedure for CLU to decide the large quantifier-free formulas. Previous attempts involving SVC [2], were unsuccessful, because of the rational interpretation of variables<sup>3</sup>. This produces numerous spurious counterexamples, since it does not properly model the integer semantics of the array indices.

## 4.5 Verifying Superscalar Processors

Previous attempts at the verification of out-of-order processors with unbounded resources consider processors that could only dispatch and retire a single instruction at each step<sup>4</sup>. Increasing the dispatch-width or the retirement width results in a more complex control logic for additional data forwarding. It also breaks the symmetry of entries in the reorder buffer, because of the explicit priority

<sup>2</sup> The proof-script size of `000.ex_br_mem` is smaller than `000.ex_br_mem_simp` because of some cleanups in the former scripts

<sup>3</sup> Private Communication with Robert Jones

<sup>4</sup> Velev [15] considers large dispatch and retire widths for processors with bounded resources. But his technique based on rewriting is very specific to the model in the paper and is hard to extend to models with even register renaming.

among the instructions in the dispatch-width. This reduces the effectiveness of approaches which use symmetry to reduce the complexity of the state space. Since our technique does not explicitly depend on symmetry, we can handle out-of-order processors with superscalar nature. To illustrate this, we generated a set of models with different dispatch and retirement widths on top of the processor model `000.base`. Unlike `000.base`, an arbitrary number of instructions can execute on any step. Moreover, dispatch, execute and retirement can occur concurrently instead of the interleaving model previously considered.

Width		max # instant	max prop-vars	Time	
Dispatch	Retire			Total	Conversion
1	1	10	439	58.69	53.90
1	2	28	682	93.56	84.98
1	4	88	1060	249.42	201.42
1	6	180	1433	470.44	362.63
1	8	304	1993	800.31	553.80
2	1	12	551	86.63	84.98
2	2	28	798	137.43	118.04
2	4	88	1152	308.55	232.46
2	6	180	1660	675.86	506.96
2	8	304	2098	1040.6	605.91

**Fig. 7. Effect of processor width on verification.** “Width” denotes the width of the processor. “max # instant” denotes the maximum number of instantiations to prove any invariant, “max prop-vars” is the maximum number of propositional variables in the boolean encoding of the formulas. The “conversion” component of the time is the time the decision procedure spends encoding a CLU formula to a boolean formula.

The verification of these superscalar processors proceeded automatically with the proof script for `000.base`. This is because the invariants express relationship between state variables and they are not affected by the change in control logic. As we see in Fig 7, the verification scales to large enough dispatch and retirement widths. It should be noticed that the number of terms to instantiate grows as we increase the width. This is because more instructions explicitly affect a single instruction and the instantiation has to account for all of these instructions. But the total time required to verify grows only linearly and can scale to larger superscalar width.

## References

1. T. Arons and A. Pnueli. A comparison of two verification methods for speculative instruction execution. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '00)*, LNCS 1785, 2000.
2. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, November 1996.
3. S. Berezin, A. Biere, E. M. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out of order microprocessor verification. In *Formal Methods in Computer-Aided Design (FMCAD '98)*, LNCS 1522. Springer-Verlag, November 1998.

4. R. S. Boyer and J. Moore. A theorem prover for a computational logic. In *10th Conference on Automated Deduction (CADE '90)*, 1990.
5. R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.
6. 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 *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, July 2002.
7. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80, June 1994.
8. Y. Gurevich. The decision problem for standard classes. *The Journal of Symbolic Logic*, 41(2):460–464, June 1976.
9. R. Hosabettu, G. Gopalakrishnan, and M. Srivas. Verifying advanced microarchitectures that support speculation and exceptions. In *(CAV 2000)*, LNCS 1855, July 2000.
10. R. Jhala and K. McMillan. Microarchitecture verification by compositional model checking. In *Computer-Aided Verification*, LNCS 2102, July 2001.
11. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159. Springer-Verlag, Nov 2002.
12. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, June 1992.
13. J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Computer-Aided Verification (CAV '98)*, LNCS 1427, June 1998.
14. J. U. Skakkebaek, R. B. Jones, and D. L. Dill. Formal verification of out-of-order execution using incremental flushing. In *(CAV '98)*, LNCS 1427, June 1998.
15. M. N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *Design, Automation and Test in Europe (DATE '02)*, pages 28–35, March 2002.

# Theorem Proving Using Lazy Proof Explication

Cormac Flanagan<sup>1</sup>, Rajeev Joshi<sup>1</sup>, Xinming Ou<sup>2</sup>, and James B. Saxe<sup>1</sup>

<sup>1</sup> Systems Research Center, HP Labs, Palo Alto, CA

<sup>2</sup> Princeton University, Princeton, NJ

**Abstract.** Many verification problems reduce to proving the validity of formulas involving both propositional connectives and domain-specific functions and predicates. This paper presents an *explicating* theorem prover architecture that leverages recent advances in propositional SAT solving and the development of proof-generating domain-specific procedures. We describe the implementation of an explicating prover based on this architecture that supports propositional logic, the theory of equality with uninterpreted function symbols, linear arithmetic, and the theory of arrays. We have applied this prover to a range of processor, cache coherence, and timed automata verification problems. We present experimental results on the performance of the prover, and on the performance impact of important design decisions in our implementation.

## 1 Introduction

In 1979, Nelson and Oppen [18] introduced a scheme for combining a collection of decision procedures for disjoint underlying theories, together with backtracking search, to obtain a theorem prover for formulas incorporating both propositional connectives and arbitrarily mixed application of functions and predicates of the various theories. In this paper, we propose a prover architecture based on a new style of interaction between propositional and theory-specific decision procedures. Unlike the traditional Nelson-Oppen method, our architecture separates the propositional search from the work done by decision procedures for the underlying theories. It thereby allows us to gain efficiency by taking advantage of recent advances in propositional SAT solving and the development of proof-generating decision procedures.

In the rest of this introduction, we illustrate the key ideas of our approach by means of an example. In later sections, we describe our architecture in more detail, report on our experience with a prototype implementation, discuss various design choices that impact performance, and compare our work with related approaches.

### 1.1 Our Approach

To simplify the exposition, we consider the problem of determining whether a given formula, called the *query*, is *satisfiable* (this is the dual of the theorem-proving problem). For instance, consider checking the satisfiability of:

$$(a = b) \wedge (\neg(f(a) = f(b)) \vee b = c) \wedge \neg(f(a) = f(c)) \quad (1)$$

Our approach uses a propositional SAT solver together with suitable decision procedures. For this example, we need only one theory-specific decision procedure, for the theory of equality with uninterpreted function symbols (EUF). A more useful prover would employ a larger collection of theories, cooperating according to the Nelson-Oppen equality-sharing protocol.

We translate the given problem into a purely propositional formula by introducing propositional variables  $v_1 \dots v_4$ , called *proxies*, as shown below:

$$\overbrace{(a = b)}^{v_1} \wedge (\neg \overbrace{(f(a) = f(b))}^{v_2}) \vee \overbrace{(b = c)}^{v_3} \wedge \neg \overbrace{(f(a) = f(c))}^{v_4} \quad (2)$$

Replacing each atomic formula in the query with the corresponding propositional proxy, we obtain the propositional formula:

$$(v_1) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_4) \quad (3)$$

We refer to the atomic formula associated with a proxy as its *interpretation*; thus  $a = b$  is the interpretation of  $v_1$ .

Formula (3) is an *abstraction* of query (1) in the sense that, given any satisfying assignment for (1) we can obtain a satisfying assignment for (3) simply by assigning to each proxy the truth value of its interpretation. Clearly, however, the converse does not hold in general, since the truth values are assigned to the proxies without considering their interpretations. Our strategy is to use the underlying theories to produce a sequence of successively stronger propositional abstractions until either (a) the propositional abstraction becomes unsatisfiable (in which case the original query was itself unsatisfiable), or (b) the abstraction remains satisfiable even when the proxies are interpreted as atomic formulas (in which case we have a satisfying assignment for the query).

We start by invoking a propositional SAT solver to solve the initial propositional abstraction (3). Suppose that our solver returns with the satisfying assignment that assigns **true** to  $v_1$  and **false** to  $v_2, v_3$  and  $v_4$ . Next, we assert the associated interpretations of these proxies to the underlying EUF theory, which detects that they are contradictory.

At this point, a conventional Nelson-Oppen prover like Simplify [11] would backtrack and search for a different satisfying assignment for (3), perhaps coming up with the assignment in which  $v_1$  and  $v_3$  are assigned **true**, while  $v_2$  and  $v_4$  are assigned **false**. This assignment would again be found inconsistent with EUF, and so on. Note, however, that  $a = b$  and  $\neg(f(a) = f(b))$  are mutually inconsistent by themselves. Thus, there is no point in considering any further assignments in which  $v_1$  is assigned **true** and  $v_2$  is assigned **false**.

To exploit this observation, we depart from the conventional approach and assume the existence of a decision procedure for EUF that, given a conjunction of inconsistent literals, is capable of producing a proof of the inconsistency. Returning to our example, when the interpretations of the proxies are asserted to the EUF procedure, it reports the inconsistency of  $a = b$  and  $\neg(f(a) = f(b))$  by *explicating* the following “proof”, which is an instance of the congruence



axiom,

$$\overbrace{a = b}^{v_1} \Rightarrow \overbrace{f(a) = f(b)}^{v_2}$$

We use this proof to refine our propositional abstraction by adding the additional clause  $(\neg v_1 \vee v_2)$ , obtaining

$$(v_1) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_4) \wedge (\neg v_1 \vee v_2) \quad (4)$$

Note that addition of this “explicated proof” to the propositional abstraction allows the SAT solver to refute the SAT assignment using purely propositional reasoning. Next, we invoke the SAT solver on (4). This time, it finds the satisfying assignment in which  $v_1, v_2, v_3$  are assigned **true**, and  $v_4$  is assigned **false**. As before, we assert the associated interpretations to the underlying EUF theory. The theory finds the assignment to be inconsistent, and explicates the following proof of inconsistency:

$$\overbrace{a = b}^{v_1} \wedge \overbrace{b = c}^{v_3} \Rightarrow \overbrace{f(a) = f(c)}^{v_4}$$

Using this proof, we refine our propositional abstraction to

$$(v_1) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_4) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_3 \vee v_4) \quad (5)$$

The SAT solver finds that (5) is now unsatisfiable, so we conclude that the original query (1) was itself unsatisfiable.

## 1.2 Motivation

The ideas described in this paper grew out of experience with the design and use of the theorem prover “Simplify” [11], which is based on the traditional Nelson-Oppen design. During our use of Simplify, we observed two serious performance problems. First, the backtracking search algorithm that we used for propositional inference had been far surpassed by recently developed fast SAT solvers [19,22,13]. Second, if the prover was in the midst of deeply nested case splits when it detected an inconsistency with an underlying theory, the backtracking search engine gained no information about which tentative truth assignments (besides the most recent) actually contributed to the inconsistency. Consequently the decision procedure was often forced repeatedly to rediscover the same inconsistency in later branches of the search, which differed only in the truth assignments to other, irrelevant atomic formulas. This led to our exploration of proof-generating decision procedures; such procedures essentially identify useful theory-specific facts, which are then projected into the propositional domain. We thus leverage the efficiency of modern SAT solvers, which can reuse the explicated clauses much more efficiently than the theory-specific decision procedure could regenerate them.

Our explicated clauses resemble the *conflict clauses* generated by modern SAT solvers such as GRASP [19], SATO [22], and Chaff [13,23], in that (i) after

being generated once they can be reused many times, and (ii) they are generated on the basis of their demonstrated utility in refuting some attempted satisfying assignment. The difference is that our explicated clauses are theory-specific logical consequences of the interpretations of the proxy variables, and they are discovered by the theory-specific decision procedures. In contrast, a SAT solver's conflict clauses are propositional consequences of the given clauses, and they are discovered by analyzing contradictions detected during boolean constraint propagation in the SAT solver.

We are by no means the only ones to notice that modern SAT solvers can be usefully integrated into the Nelson-Oppen design. Similar ideas have been recently proposed by Barrett, Dill and Stump [4] and by de Moura and Rueß [10]. Our approach differs from their work in a number of critical design decisions. This paper discusses these design decisions and evaluates their performance impact.

## 2 Architecture

### 2.1 Terminology

We use terminology that is standard in the literature. A *term* is a variable or an application of a function to terms. Thus,  $x$ ,  $x + 3$  and  $f(x, y)$  are all terms. An *atomic formula* is a propositional variable or an application of a predicate symbol to some terms. Thus,  $x + 3 < 5$  and  $f(x) = f(y)$  are atomic formulae. A *literal* is either an atomic formula or its negation, and a *clause* is a disjunction of literals. A *monome* is a conjunction of literals in which no atomic formula is both affirmed and negated. We identify a monome  $M$  with the partial truth assignment that assigns **true** to atomic formulae that are conjuncts of  $M$  and assigns **false** to atomic formulae whose negations are conjuncts of  $M$ .

The task of a satisfier is to decide whether an input formula, called a *query*, is satisfiable for a given set of underlying theories. The underlying theories may assume particular semantics for some predicate and function symbols, such as  $>$  and  $+$ , while leaving others uninterpreted. A *satisfying assignment* for a query is a monome that is consistent with the underlying theories of the satisfier and entails the query by propositional inference alone (i.e., treating all syntactically distinct atomic formulae as if they were distinct propositional variables).

### 2.2 Architecture

Figure 1 sketches the main algorithm for our satisfier. As shown, given a query  $F$ , we introduce proxies for the atomic formulae of  $F$ , along with a mapping  $\Gamma$  relating these proxies to the atomic formulae. This results in the SAT problem  $\Gamma^{-1}(F)$ . We invoke the SAT solver on  $\Gamma^{-1}(F)$  by invoking *SAT-solve*, which is expected to return either **null** (if the given SAT problem is unsatisfiable) or a satisfying *TruthAssignment*  $A$ . If *SAT-solve* returns **null**, it means  $\Gamma^{-1}(F)$  is unsatisfiable, so we can deduce that  $F$  itself was unsatisfiable, and we are done. Otherwise, we use the satisfying assignment  $A$  provided by the SAT solver, and

```

Input: A query  $F$ 
Output: A monome satisfying  $F$ , or null, indicating that  $F$  is unsatisfiable

function satisfy(Formula  $F$ ) : Monome {
  while (true) {
    allocate proxy propositional variables for atomic formulae in  $F$ , and
    create mapping  $\Gamma$  from proxies to atomic formulae;
    TruthAssignment  $A := SAT\text{-}solve(\Gamma^{-1}(F))$ ;
    if ( $A = \text{null}$ ) {  $\Gamma^{-1}(F)$  is unsatisfiable, hence so is  $F$ 
      return null;
    } else {
      Monome  $M := \Gamma(A)$ ;
      Formula  $E := check(M)$ ;
      if ( $E = \text{null}$ ) {  $E$  is satisfiable, and so is  $F$ 
        return  $\Gamma(A)$ ;
      } else { decision procedure found  $M$  inconsistent and explicated  $E$ 
         $F := F \wedge E$ ;
      }
    }
  }
}

```

**Fig. 1.** A satisfiability algorithm using proof explication

invoke the procedure *check* which determines if the monome  $M$  is consistent with the underlying theories. If *check* returns **null**, it means that  $M$  is consistent with the underlying theories, so we have obtained a satisfying assignment to our original  $F$  and we are done. Otherwise, *check* determines that  $M$  is inconsistent with the underlying theories, and returns a proof  $E$  of the inconsistency. We update the query  $F$  by conjoining  $E$  to it, and continue by mapping the query using  $\Gamma$  and reinvoking the SAT solver as described above. We assume that  $E$  is (1) entailed by the axioms of the underlying theories, and (2) propositionally entails the negation of the given monome  $M$ . Condition (2) suffices to show that the algorithm terminates, since it ensures that at each step, we strictly strengthen the query. Condition (1) suffices to show soundness, so that if the updated query becomes propositionally unsatisfiable, we may conclude that the original query was unsatisfiable. Thus, we can view the iterative process as transferring information from the underlying theories into the propositional domain. Eventually we either strengthen the query until it becomes propositionally unsatisfiable, or the SAT solver finds a satisfying assignment whose interpretation is consistent with the underlying theories.

### 3 Implementation and Evaluation

To explore the performance benefits of generating explicated clauses, we implemented a satisfier, called **Verifun**, based on the architecture of Figure 1. Verifun

consists of approximately 10,500 lines of Java and around 800 lines of C code. The bulk of the Java code implements explicating decision procedures for EUF, rational linear arithmetic, and the theory of arrays. The decision procedure for EUF is based on the *E-graph* data structure proposed by Nelson and Oppen [17], which we adapted to explicate proofs of transitivity and congruence. The decision procedure for linear arithmetic is based on a variation [16] of the Simplex algorithm that we have modified to support proof explication. Finally, the decision procedure for arrays uses pattern matching to produce ground instances of select and store axioms. The C code implements the interface to the SAT solver.

The Verifun implementation extends the basic explicating architecture of Figure 1 with a number of improvements, which we describe below.

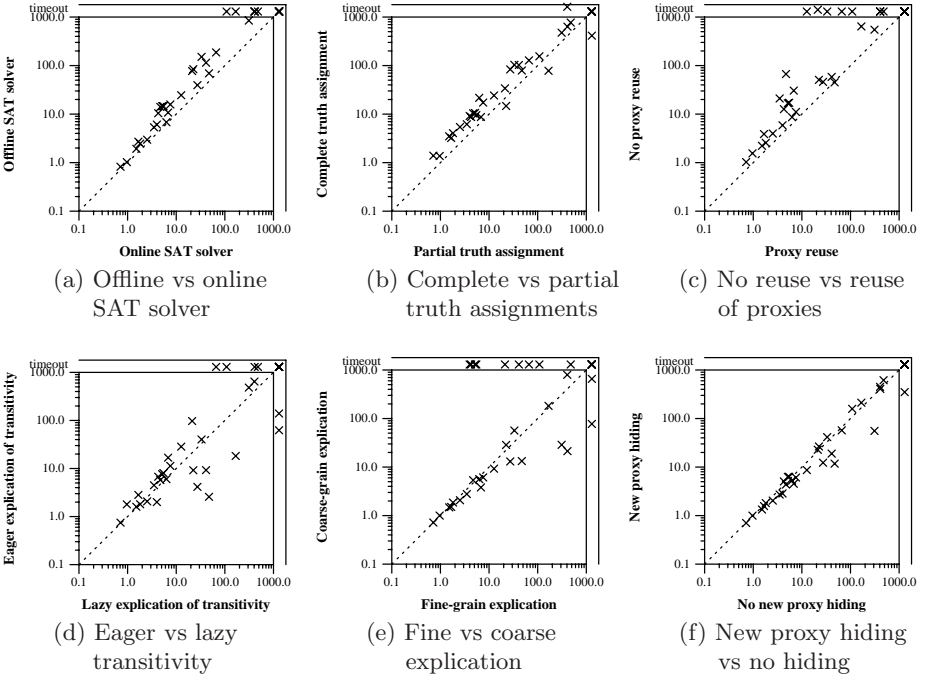
### 3.1 Online vs. Offline Propositional SAT Solving

The architecture of Figure 1 uses an *offline* SAT solver, which is reinvoked from scratch each time the SAT problem is extended with additional explicated clauses. Each reinvocation is likely to repeat much of the work of the previous invocation. To avoid this performance bottleneck, Verifun uses a customized *online* variant of the zChaff SAT solver [13]. After reporting a satisfying assignment, this online SAT solver accepts a set of additional clauses and then continues its backtracking search from the point at which that assignment was found, and thus avoids reexamining the portion of the search space that has already been refuted.

To illustrate the benefit of online SAT solving, Figure 2(a) compares the performance of two versions of Verifun, which use online and offline versions of zChaff, respectively. We used a benchmark suite of 38 processor and cache coherence verification problems provided by the UCLID group at CMU [6]. These problems are expressed in a logic that includes equality, uninterpreted functions, simple counter arithmetic (increment and decrement operations), and the usual ordering over the integers. All experiments in this paper were performed on a machine with dual 1GHz Pentium III processors and 1GB of RAM, running Redhat Linux 7.1. Since Verifun is single-threaded, it uses just one of the two processors. We consider an invocation of the prover to *timeout* if it took more than 1000 seconds, ran out of memory, or otherwise crashed. As expected, the online SAT solver significantly improves the performance of Verifun and enables it to terminate on more of the benchmark problems. Note that the 'x' in the top right of this graph (and in subsequent graphs) covers several benchmarks that timed out under both Verifun configurations.

### 3.2 Partial vs. Complete Truth Assignments

As described in Section 2.2, the SAT solution  $A$  is converted to a monome  $M$  that entails the query by propositional reasoning (although  $M$  may not be consistent with the underlying theories). By default,  $M$  is *complete*, in that it associates a truth value with every atomic formula in the query. An important optimization is to compute from  $M$  a minimal sub-monome  $M'$  that still entails the query.



**Fig. 2.** Scattergraph of runtime (in seconds) comparing versions of Verifun on the UCLID benchmarks. Except where labeled otherwise, Verifun used the online SAT solver, partial truth assignments, proxy reuse, lazy transitivity, fine-grained explication, and no hiding of new proxy variables.

Since any monome extending  $M'$  also entails the query,  $M$  is an extension of  $M'$  that assigns arbitrary truth values to atomic formulas not mentioned in  $M'$ , which in turn may cause *check* to explicate clauses that are not very useful. Therefore, we instead apply *check* to the partial monome or truth assignment  $M'$ . Figure 2(b) illustrates the benefit of this optimization.

### 3.3 Proxy Reuse

Since standard SAT solvers require their input to be in conjunctive normal form (CNF), Verifun first converts the given query to CNF. To avoid exponential blow-up, Verifun introduces additional *proxy* variables for certain subformulas in the query, as necessary. If a particular subformula appears multiple times in the query, an important optimization is to reuse the same proxy variable for that subformula, instead of introducing a new proxy variable for each occurrence. Figure 2(c) illustrates the substantial performance benefit of this optimization.

### 3.4 Eager Transitivity

By default, Verifun explicates clauses in a *lazy* manner, in response to satisfying assignments produced by the SAT solver. An alternative approach proposed by

Velev [8] and others [6,7,12] is to perform this explication *eagerly*, before running the SAT solver. The relative performance of the two approaches is unclear, in part because lazy explication generates fewer clauses, but invokes the SAT solver multiple times.

As a first step in comparing the two approaches, we extended Verifun to perform eager explication of clauses related to transitivity of equality. These clauses are generated by maintaining a graph whose vertices are the set of all terms, and whose edges are the set of all equalities that appear (negated or not) in the current query. At each step, before the SAT solver is invoked, we add edges to make the graph chordal, using the well-known linear-time algorithm of Tarjan and Yannakakis [21]. Next, we enumerate all triangles in this graph that contain at least one newly added edge. For each such triangle, we generate the three possible instances of the transitivity axiom. Figure 2(d) illustrates the effect of eager explication on Verifun’s running time on the UCLID benchmarks. Interestingly, although eager explication significantly reduces the number of iterations though the main loop of Verifun, often by an order of magnitude, the timing results indicate that eager explication does not produce a consistent improvement in Verifun’s performance over lazy explication.

### 3.5 Granularity of Explication

When the *check* decision procedure detects an inconsistency, there is generally a choice about which clauses to explicate. To illustrate this idea, suppose the decision procedure is given the following inconsistent collection of literals:

$$a = b, b = c, f(a) \neq f(c)$$

When the decision procedure detects this inconsistency, it could follow the *coarse-grain* strategy of explicating this inconsistency using the single clause:

$$a = b \wedge b = c \Rightarrow f(a) = f(c)$$

A second strategy is *fine-grain* explication, whereby the decision procedure explicates the proof of inconsistency using separate instances of the transitivity and congruence axioms:

$$\begin{aligned} a = b \wedge b = c &\Rightarrow a = c \\ a = c &\Rightarrow f(a) = f(c) \end{aligned}$$

Fine-grained explication produces more and smaller explicated clauses than coarse-grained explication. This is likely to result in the SAT solver doing more unit propagation, but may allow the SAT solver to refute more of the search space without reinvoking the decision procedure. In the example above, the clause  $a = c \Rightarrow f(a) = f(c)$  might help prune a part of the search space where  $a = c$  holds, even if the transitivity chain  $a = b = c$  does not hold. By comparison, the coarse-grained explicated clause would not have been useful.

Figure 2(e) compares the performance of coarse-grained versus fine-grained explication. On several benchmarks, the coarse-grained strategy times out because it generates a very large number of clauses, where each clause is very specific and only refutes a small portion of the search space. Fine-grained explication terminates more often, but is sometimes slower when it does terminate.

We conjecture this slowdown is because fine-grained explication produces clauses containing atomic formulas (such as  $a = c$  in the example above) that do not occur in the original query, which causes the SAT solver to assign truth values to these new atomic formulas. Thus, subsequent explication may be necessary to refute inconsistent assignments to the new atomic formulas.

To avoid this problem, we extended Verifun to hide the truth assignment to these new atomic formulas from the decision procedure. In particular, when the SAT solver returns a satisfying assignment, Verifun passes to the decision procedure only the truth assignments for the original atomic formulas, and not for the new atomic formulas. Figure 2(f) shows that this *hiding new proxies* strategy produces a significant performance improvement, without introducing the problems associated with the coarse-grained strategy.

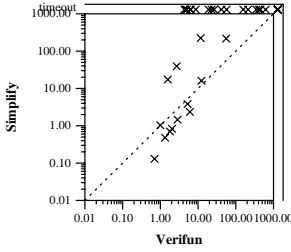
### 3.6 Comparison to Other Theorem Provers

We next compare the performance of Verifun with three comparable provers. Figure 3(a) compares Verifun with the Simplify theorem prover [11] on the UCLID benchmarks, and shows that Verifun scales much better to large problems. In several cases, Verifun is more than two orders of magnitude faster than Simplify, due to its use of explicated clauses and a fast SAT solver.

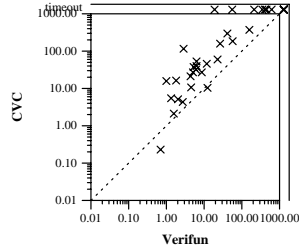
Figure 3(b) compares Verifun to the Cooperating Validity Checker (CVC) [4] on the UCLID benchmarks. The results show that Verifun performs better than CVC on these benchmarks, perhaps because CVC uses coarse-grained explication, which our experiments suggest is worse than Verifun's fine-grained explication.

Figure 3(c) and (d) compare Verifun with the Stanford Validity Checker (SVC) [3]. Figure 3(c) uses the UCLID benchmarks plus an additional benchmark provided by Velev that uses EUF and the theory of arrays. Figure 2 (d) uses the Math-SAT postoffice suite<sup>1</sup> of 41 timed automata verification problems [2,1]. Interestingly, SVC performs better than Verifun on the UCLID benchmarks, but worse on the postoffice benchmarks, perhaps because these tools have been tuned for different problem domains. In addition, SVC is a relatively mature and stable prover written in C, whereas Verifun is a prototype written in Java, and still has significant opportunities for further optimization. For example, our decision procedures are currently non-backtracking and therefore repeat work across invocations. In most cases, a large percentage of Verifun's total running time is spent inside these decision procedures. We expect that backtracking decision procedures would significantly improve Verifun's performance.

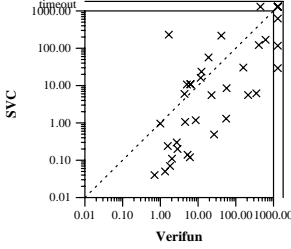
<sup>1</sup> We have not been able to run either Simplify or CVC on the postoffice benchmarks, due to incompatible formats.



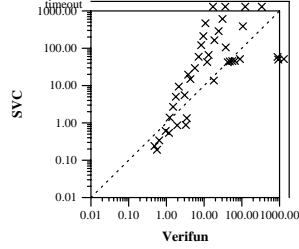
(a) Verifun vs Simplify on UCLID



(b) Verifun vs CVC on UCLID



(c) Verifun vs SVC on UCLID and Velev



(d) Verifun vs SVC on postoffice

**Fig. 3.** Scattergraph of runtime (in seconds) comparing Verifun with (a) Simplify, (b) CVC, and (c) SVC on the UCLID benchmarks, and (d) comparing Verifun with SVC on the postoffice benchmarks. Verifun used the online SAT solver, lazy transitivity, fine-grained explication, new proxy hiding, partial truth assignments, and proxy reuse.

Math-SAT [2,1] performs extremely well on the postoffice benchmarks, partly because these benchmarks include hints that Math-SAT uses to direct the search performed by the SAT solver. Verifun cannot currently exploit such hints, but its performance is superior to Math-SAT if these hints are not provided. In particular, on a 700MHz Pentium III with 1GB of RAM, Math-SAT is unable to solve any of the 5 largest problems within an hour, whereas Verifun can solve the largest one in 26 minutes.

## 4 Related Work

The idea of theorem proving by solving a sequence of incrementally growing SAT problems occurs as early as the 1960 Davis and Putnam paper [9]. However their algorithm simply enumerated all instantiations of universally quantified formulas in increasing order of some complexity measure, testing ever larger sets of instantiations for propositional consistency with the initial query. While their prover was, in principle, complete for first-order predicate calculus, it was unlikely to complete any proof requiring quantified variables to be instantiated with large terms before getting overwhelmed with numerous simpler but irrelevant instances.

The idea of adding support for proof explication to decision procedures has been explored by George Necula in the context of his work on “proof-carrying-



code” (PCC) [14,15]. However, in PCC, proof-generation is used with a different purpose, viz., to allow code receivers to verify the correctness of code with respect to a safety policy. Our concerns, on the other hand, are different: we are interested in proof-generation in order to produce a sufficient set of clauses to rule out satisfying assignments that are inconsistent with the underlying theory. In particular, the quality of the explicated proofs is not as crucial in the context of PCC.

More similar in nature to our work is the Cooperating Validity Checker (CVC) [4]. CVC also uses explicated clauses in order to control the boolean search. However, the CVC approach differs from Verifun’s in some crucial ways:

- CVC invokes its decision procedures incrementally as the SAT solver assigns truth values to propositional variables.
- Explication in CVC is coarser-grained than in Verifun.
- CVC generates proofs as new facts are inferred. On the other hand, our prover merely records sufficient information in the data structures so that proofs can be generated if needed.

Recent work by de Moura and Rueß [10] is closely related to ours, in that they too propose an architecture in which the decision procedures are invoked *lazily*, after the SAT solver has produced a complete satisfying assignment. They note the impact of proof-explicating decision procedures in pruning the search space more quickly. They also contrast the lazy decision procedure invocation approach with the eager invocation approach of provers like CVC. Our work differs from theirs in that we have focused on studying the performance impact of various design choices within the space of lazy invocation of decision procedures.

Another system employing a form of lazy explication is Math-SAT [2]. This system is specialized to the theory of linear arithmetic, for which it incorporates not only a full decision procedure but three partial decision procedures. Each decision procedure is invoked only when all weaker ones have failed to refute a potential satisfying assignment.

Verifun produces propositional projections of theory-specific facts lazily, on the basis of its actual use of those facts in refuting proposed satisfying assignments. An alternative approach is to identify at the outset and project to the propositional domain all theory-specific facts that might possibly be needed for testing a particular query. This “eager” approach has been applied by Bryant, German, and Velev [5] to a logic of EUF and arrays, and extended by Bryant, Lahiri, and Seshia [7] to include counter arithmetic. Strichman [20] has investigated the eager projection to SAT for Presburger and linear arithmetic. When employing the eager approach it is important not to explicate the exact theory-specific constraint on the atomic formulas in the query, but to identify a set of theory-specific facts guaranteed to be sufficient for deciding a query without being excessively large [8]. Where this has been possible, the eager approach has been impressively successful. For richer theories (in particular for problems involving quantification), it is unclear whether it will be possible to identify and project all necessary theory-specific facts at the outset without also including irrelevant facts that swamp the SAT solver.

## 5 Conclusion

Our experience suggests that lazy explication is a promising strategy to harness recent developments in SAT solving and proof-generating decision procedures. Our comparisons of Verifun and Simplify indicate that this approach is more efficient than the traditional Nelson-Oppen approach. Comparisons with other approaches like SVC, though promising (as shown in Figure 3(c)), are not as conclusive. This is partly because several obvious optimizations (such as back-tracking theories) are not yet implemented in Verifun.

One advantage of our approach over that of CVC is that there is less dependence on the SAT solver, which makes it easier to replace that SAT solver with the current world champion. A potential advantage of lazy explication is that it is easier to extend to additional theories than the eager explication approaches of Bryant et al. and Strichman. In particular, we have recently extended our implementation to handle quantified formulas. By comparison, it is unclear how to extend eager explication to handle quantification.

## Acknowledgments

We are grateful to Sanjit Seshia for providing us with the UCLID benchmarks in SVC and CVC formats, to Alessandro Cimatti for helping us understand the format of the Math-SAT postoffice problems, and to Rustan Leino for helpful comments on this paper.

## References

1. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Input files for Math-SAT case studies. <http://www.dit.unitn.it/~rseba/Mathsat.html>.
2. Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Proceedings of the 18th Conference on Automated Deduction*, July 2002.
3. Clark W. Barrett, David L. Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Proceedings of Formal Methods In Computer-Aided Design*, pages 187–201, November 1996.
4. Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer, July 2002.
5. Randal E. Bryant, Steven German, and Miroslav N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Proceedings 11th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 470–482. Springer, July 1999.
6. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Deciding CLU logic formulas via Boolean and pseudo-Boolean encodings. In *Proceedings of the First International Workshop on Constraints in Formal Verification*, September 2002.

7. Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer, July 2002.
8. Randal E. Bryant and Miroslav N. Velev. Boolean satisfiability with transitivity constraints. In *Proceedings 12th International Conference on Computer Aided Verification*, pages 85–98, July 2000.
9. M. Davis and H. Putnam. A computing procedure for quantification theory. *JACM*, 7:201–215, 1960.
10. Leonardo de Moura and Harald Ruess. Lemmas on demand for satisfiability solvers. In *Proceedings of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002.
11. David Detlefs, Greg Nelson, and James B. Saxe. A theorem-prover for program checking. Technical report, HP Systems Research Center, 2003. In preparation.
12. Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159. Springer, November 2002.
13. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, June 2001.
14. George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie-Mellon University, 1998. Also available as CMU Computer Science Technical Report CMU-CS-98-154.
15. George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 25–44, June 2000.
16. Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1979. A revised version of this thesis was published as Xerox PARC Computer Science Laboratory Research Report CSL-81-10.
17. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2), October 1979.
18. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, October 1979.
19. João Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5), May 1999.
20. Ofer Strichman. On solving Presburger and linear arithmetic with SAT. In *Proceedings Formal Methods in Computer-Aided Design*, pages 160–170, 2002.
21. Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13(3):566–579, August 1984.
22. Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, 1997.
23. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, November 2001.

# Enhanced Vacuity Detection in Linear Temporal Logic

Roy Armoni<sup>1</sup>, Limor Fix<sup>1</sup>, Alon Flaisher<sup>1,2</sup>, Orna Grumberg<sup>2</sup>, Nir Piterman<sup>1</sup>,  
Andreas Tiemeyer<sup>1</sup>, and Moshe Y. Vardi<sup>4\*</sup>

<sup>1</sup> Intel Design Center, Haifa.

(roy.armoni, limor.fix, alon.flaischer)@intel.com

(nir.piterman, andreas.tiemeyer)@intel.com

<sup>2</sup> Technion, Israel Institute of Technology. orna@cs.technion.ac.il

<sup>3</sup> Rice University. vardi@cs.rice.edu

**Abstract.** One of the advantages of temporal-logic model-checking tools is their ability to accompany a negative answer to a correctness query with a counterexample to the satisfaction of the specification in the system. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no witness for the satisfaction of the specification. In the last few years there has been growing awareness of the importance of suspecting the system or the specification of containing an error also in cases where model checking succeeds. In particular, several works have recently focused on the detection of the *vacuous satisfaction* of temporal logic specifications. For example, when verifying a system with respect to the specification  $\varphi = G(req \rightarrow F grant)$  (“every request is eventually followed by a grant”), we say that  $\varphi$  is satisfied vacuously in systems in which requests are never sent. Current works have focused on detecting vacuity with respect to subformula occurrences. In this work we investigate vacuity detection with respect to subformulas with multiple occurrences.

The generality of our framework requires us to re-examine the basic intuition underlying the concept of vacuity, which until now has been defined as sensitivity with respect to syntactic perturbation. We study sensitivity with respect to semantic perturbation, which we model by universal propositional quantification. We show that this yields a hierarchy of vacuity notions. We argue that the right notion is that of vacuity defined with respect to traces. We then provide an algorithm for vacuity detection and discuss pragmatic aspects.

## 1 Introduction

*Temporal logics*, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying concurrent systems [Pnu77]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* systems [CE81, CES86, LP85, QS81, VW86]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state systems, as well as from the great ease of use of fully algorithmic methods. In

---

\* Supported in part by NSF grants CCR-9988322, IIS-9908435, IIS-9978135, and EIA-0086264, by BSF grant 9800096, and by a grant from the Intel Corporation.

temporal-logic *model checking*, we verify the correctness of a finite-state system with respect to a desired behavior by checking whether a labeled state-transition graph that models the system satisfies a temporal logic formula that specifies this behavior (for an in-depth survey, see [CGP99]).

Beyond being fully-automatic, an additional attraction of model-checking tools is their ability to accompany a negative answer to the correctness query with a counterexample to the satisfaction of the specification in the system. Thus, together with a negative answer, the model checker returns some erroneous execution of the system. These counterexamples are very important and can be essential in detecting subtle errors in complex designs [CGMZ95]. On the other hand, when the answer to the correctness query is positive, most model-checking tools provide no witness for the satisfaction of the specification in the system. Since a positive answer means that the system is correct with respect to the specification, this may, a priori, seem like a reasonable policy. In the last few years, however, industrial practitioners have become increasingly aware of the importance of checking the validity of a positive result of model checking. The main justification for suspecting the validity of a positive result is the possibility of errors in the modeling of the system or of the desired behavior, i.e., the specification.

Early work on “suspecting a positive answer” concerns the fact that temporal logic formulas can suffer from *antecedent failure* [BB94]. For example, in verifying a system with respect to the CTL specification  $\varphi = AG(req \rightarrow AF grant)$  (“every request is eventually followed by a grant”), one should distinguish between *vacuous satisfaction* of  $\varphi$ , which is immediate in systems in which requests are never sent, and non-vacuous satisfaction, in systems where requests are sometimes sent. Evidently, vacuous satisfaction suggests some unexpected properties of the system, namely the absence of behaviors in which the antecedent of  $\varphi$  is satisfied.

Several years of practical experience in formal verification have convinced the verification group at the IBM Haifa Research Laboratory that vacuity is a serious problem [BBER97]. To quote from [BBER97]: “Our experience has shown that typically 20% of specifications pass vacuously during the first formal-verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment.” The usefulness of vacuity analysis is also demonstrated via several case studies in [PS02]. Often, it is possible to detect vacuity easily by checking the system with respect to hand-written formulas that ensure the satisfaction of the preconditions in the specification [BB94, PP95]. To the best of our knowledge, this rather unsystematic approach is the prevailing one in the industry for dealing with vacuous satisfaction. For example, the FormalCheck tool [Kur98] uses “sanity checks”, which include a search for triggering conditions that are never enabled.

These observations led Beer et al. to develop a method for automatic testing of vacuity [BBER97]. Vacuity is defined as follows: a formula  $\varphi$  is satisfied in a system  $M$  vacuously if it is satisfied in  $M$ , but some subformula  $\psi$  of  $\varphi$  does not *affect*  $\varphi$  in  $M$ , which means that  $M$  also satisfies  $\varphi[\psi \leftarrow \psi']$  for all formulas  $\psi'$  (here,  $\varphi[\psi \leftarrow \psi']$  denotes the result of substituting  $\psi'$  for  $\psi$  in  $\varphi$ ). Beer et al. proposed testing vacuity by means of *witness formulas*. Formally, we say that a formula  $\varphi'$  is a *witness formula* for the specification  $\varphi$  if a system  $M$  satisfies  $\varphi$  non-vacuously iff  $M$  satisfies both  $\varphi$  and  $\varphi'$ . In the example above, it is not hard to see that a system satisfies  $\varphi$  non-vacuously iff

it also satisfies *EFreq*. In general, however, the generation of witness formulas is not trivial, especially when we are interested in other types of vacuity passes, which are more complex than antecedent failure. While [BBER97] nicely set the basis for a methodology for detecting vacuity in temporal-logic specifications, the particular method described in [BBER97] is quite limited.

A general method for detection of vacuity for specifications in CTL\* (and hence also LTL, which was not handled by [BBER97]) was presented in [KV99,KV03]. The key idea there is a general method for generating witness formulas. It is shown in [KV03] that instead of replacing a subformula  $\psi$  by all subformulas  $\psi'$ , it suffices to replace it by either **true** or **false** depending on whether  $\psi$  occurs in  $\varphi$  with negative polarity (i.e., under an odd number of negations) or positive polarity (i.e., under an even number of negations). Thus, vacuity checking amounts to model checking witness formulas with respect to all (or some) of the subformulas of the specification  $\varphi$ . It is important to note that the method in [KV03] is for vacuity with respect to subformula occurrences. The key feature of occurrences is that a subformula occurrence has a *pure* polarity (exclusively negative or positive). In fact, it is shown in [KV03] that the method is not applicable to subformulas with mixed polarity (both negative and positive occurrences).

Recent experience with industrial-strength property-specification languages such as ForSpec [AFF<sup>+</sup>02] suggests that the restriction to subformula occurrences of pure polarity is not negligible. ForSpec, which is a linear-time language, is significantly richer syntactically (and semantically) than LTL. It has a rich set of arithmetical and Boolean operators. As a result, even subformula occurrences may not have pure polarity, e.g., in the formulas  $p \oplus q$  ( $\oplus$  denotes exclusive or). While we can rewrite  $p \oplus q$  as  $(p \wedge \neg q) \vee (\neg p \wedge q)$ , it forces the user to think of every subformula occurrence of mixed polarity as two distinct occurrences, which is rather unnatural. Also, a subformula may occur in the specification multiple times, so it need not have a pure polarity even if each occurrence has a pure polarity. For example, if the LTL formula  $G(p \rightarrow p)$  holds in a system  $M$  then we'd expect it to hold vacuously with respect to the subformula  $p$  (which has a mixed polarity), though not necessarily with respect to either occurrence of  $p$ , because both formulas  $G(\mathbf{true} \rightarrow p)$  and  $G(p \rightarrow \mathbf{false})$  may fail in  $M$ . (Surely, the fact that  $G(\mathbf{true} \rightarrow \mathbf{false})$  fails in  $M$  should not entail that  $G(p \rightarrow p)$  holds in  $M$  non-vacuously.) Our goal is to remove the restriction in [KV03] to subformula occurrences of pure polarity. To keep things simple, we stick to LTL and consider vacuity with respect to subformulas, rather than with respect to subformula occurrences. We comment on the extension of our framework to ForSpec at the end of the paper.

The generality of our framework requires us to re-examine the basic intuition underlying the concept of vacuity, which is that a formula  $\varphi$  is satisfied in a system  $M$  vacuously if it is satisfied in  $M$  but some subformula  $\psi$  of  $\varphi$  does not *affect*  $\varphi$  in  $M$ . It is less clear, however, what does “does not affect” mean. Intuitively, it means that we can “perturb”  $\psi$  without affecting the truth of  $\varphi$  in  $M$ . Both [BBER97] and [KV03] consider only syntactic perturbation, but no justification is offered for this decision. We argue that another notion to consider is that of semantic perturbation, where the *truth value* of  $\psi$  in  $M$  is perturbed arbitrarily. The first part of the paper is an examination in depth of this approach. We model arbitrary semantic perturbation by a universal quantifier, which in turn is open to two interpretations (cf. [Kup95]). It turns out that we get

two notions of “does not affect” (and therefore also of vacuity), depending on whether universal quantification is interpreted with respect to the system  $M$  or with respect to its set of computations. We refer to these two semantics as “structure semantics” and “trace semantics”. Interestingly, the original, syntactic, notion of perturbation falls between the two semantic notions.

We argue then that trace semantics is the preferred one for vacuity checking. Structure semantics is simply too weak, yielding vacuity too easily. Formula semantics is more discriminating, but it is not robust, depending too much on the syntax of the language. In addition, these two semantics yield notions of vacuity that are computationally intractable. In contrast, trace semantics is not only intuitive and robust, but it can be checked easily by a model checker.

In the final part of the paper we address several pragmatic aspects of vacuity checking. We first discuss whether vacuity should be checked with respect to subformulas or subformula occurrences and argue that both checks are necessary. We then discuss how the number of vacuity checks can be minimized. We also discuss how vacuity results should be reported to the user. Finally, we describe our experience of implementing vacuity checking in the context of a ForSpec-based model checker.

A version with full proofs can be downloaded from the authors’ homepages.

## 2 Preliminaries

**LTL.** Formulas of LTL are built from a set  $AP$  of atomic propositions using the usual Boolean operators and the temporal operators  $X$  (“next time”) and  $U$  (“until”). Given a set  $AP$ , an LTL formula is:

- **true**, **false**,  $p$  for  $p \in AP$ .
- $\neg\psi$ ,  $\psi \wedge \varphi$ ,  $X\psi$ , or  $\psi U \varphi$ , where  $\psi$  and  $\varphi$  are LTL formulas.

We define satisfaction of LTL formulas with respect to computations of Kripke structures. A Kripke structure is  $M = \langle AP, S, S_0, R, L \rangle$  where  $AP$  is the set of atomic propositions,  $S$  is a set of states,  $S_0$  is a set of initial states,  $R \subseteq S \times S$  is a total transition relation, and  $L : AP \rightarrow 2^S$  assigns to each atomic proposition the set of states in which it holds. A *computation* is a sequence of states  $\pi = s_0, s_1, \dots$  such that  $s_0 \in S_0$  and for all  $i \geq 0$  we have  $(s_i, s_{i+1}) \in R$ . We denote the set of computations of  $M$  by  $\mathcal{T}(M)$  and the suffix  $s_j, s_{j+1}, \dots$  of  $\pi$  by  $\pi^j$ .

The semantics of LTL is defined with respect to computations and locations. We denote  $M, \pi, i \models \varphi$  when the LTL formula  $\varphi$  holds in the computation  $\pi$  at location  $i \geq 0$ . A computation  $\pi$  satisfies an LTL formula  $\varphi$ , denoted  $\pi \models \varphi$  if  $\pi, 0 \models \varphi$ . The structure  $M$  satisfies  $\varphi$ , denoted  $M \models \varphi$  if for every computation  $\pi \in \mathcal{T}(M)$  we have  $\pi \models \varphi$ . For a full definition of the semantics of LTL we refer the reader to [Eme90].

An occurrence of formula  $\psi$  of  $\varphi$  is of *positive polarity* in  $\varphi$  if it is in the scope of an even number of negations, and of *negative polarity* otherwise. The polarity of a subformula is defined by the polarity of its occurrences as follows. Formula  $\psi$  is of *positive polarity* if all occurrences of  $\psi$  in  $\varphi$  are of positive polarity, of *negative polarity* if all occurrences of  $\psi$  in  $\varphi$  are of negative polarity, of *pure polarity* if it is either of positive or negative polarity, and of *mixed polarity* otherwise.

Given a formula  $\varphi$  and a subformula of pure polarity  $\psi$  we denote by  $\varphi[\psi \leftarrow \perp]$  the formula obtained from  $\varphi$  by replacing  $\psi$  by **true** (**false**) if  $\psi$  is of negative (positive) polarity.

**UQLTL.** The logic UQLTL augments LTL with universal quantification over *propositional variables*. Let  $X$  be a set of propositional variables. The syntax of UQLTL is as follows. If  $\varphi$  is an LTL formula over the set of atomic propositions  $AP \cup X$ , then  $\forall X \varphi$  is a UQLTL formula. E.g.,  $\forall x G(x \rightarrow p)$  is a legal UQLTL formula, while  $G \forall x (x \rightarrow p)$  is not. UQLTL is a subset of *Quantified Propositional Temporal Logic* [SVW85], where the free variables are quantified universally. We use  $x$  to denote a propositional variable. A *closed* formula is a formula with no free propositional variables.

We now define two semantics for UQLTL. In *structure semantics* a propositional variable is bound to a subset of the states of the Kripke structure. In *trace semantics* a propositional variable is bound to a subset of the locations on the trace.

Let  $M$  be a Kripke structure with a set of states  $S$ , let  $\pi \in \mathcal{T}(M)$ , and let  $X$  be a set of propositional variables. A *structure assignment*  $\sigma : X \rightarrow 2^S$  maps every propositional variable  $x \in X$  to a set of states in  $S$ . We use  $s_i$  to denote the  $i$ th state along  $\pi$ , and  $\varphi$  to denote UQLTL formulas.

**Definition 1 (Structure Semantics).** *The relation  $\models_s$  is defined inductively as follows:*

- $M, \pi, i, \sigma \models_s x$  iff  $s_i \in \sigma(x)$ .
- $M, \pi, i, \sigma \models_s \forall x \varphi$  iff  $M, \pi, i, \sigma[x \leftarrow S'] \models_s \varphi$  for every  $S' \subseteq S$ .
- For other formulas,  $M, \pi, i, \sigma \models_s$  is defined as in LTL.

We now define the trace semantics for UQLTL. Let  $X$  be a set of propositional variables. A *trace assignment*  $\alpha : X \rightarrow 2^{\mathbb{N}}$  maps a propositional variable  $x \in X$  to a set of natural numbers (points on a path).

**Definition 2 (Trace Semantics).** *The relation  $\models_t$  is defined inductively as follows:*

- $M, \pi, i, \alpha \models_t x$  iff  $i \in \alpha(x)$ .
- $M, \pi, i, \alpha \models_t \forall x \varphi$  iff  $M, \pi, i, \alpha[x \leftarrow N'] \models_t \varphi$  for every  $N' \subseteq \mathbb{N}$ .
- For other formulas,  $M, \pi, i, \sigma \models_t$  is defined as in LTL.

A closed UQLTL formula  $\varphi$  is *structure satisfied* at point  $i$  of trace  $\pi \in \mathcal{T}(M)$ , denoted  $M, \pi, i \models_s \varphi$ , iff  $M, \pi, i, \sigma \models_s \varphi$  for some  $\sigma$  (choice is not relevant since  $\varphi$  is closed). A closed UQLTL formula  $\varphi$  is *structure satisfied* in structure  $M$ , denoted  $M \models_s \varphi$ , iff  $M, \pi, 0 \models_s \varphi$  for every trace  $\pi \in \mathcal{T}(M)$ . *Trace satisfaction* is defined similarly for a trace and for structure, and is denoted by  $\models_t$ .

Trace semantics is stronger than structure semantics in the following sense.

**Theorem 1.** *Given a structure  $M$  and a UQLTL formula  $\varphi$ ,  $M \models_t \varphi$  implies  $M \models_s \varphi$ . The reverse implication does not hold.*

The proof resembles the proofs in [Kup95] for the dual logic EQCTL. Kupferman shows that a structure might not satisfy a formula, while its computation tree does. Indeed, a trace assignment can assign a variable different values when the computation visits the same state of  $M$ . We observe that for LTL formulas both semantics are identical. That is, if  $\varphi$  is an LTL formula, then  $M \models_s \varphi$  iff  $M \models_t \varphi$ . We use  $\models$  to denote the satisfaction of LTL formulas, rather than  $\models_s$  or  $\models_t$ .



### 3 Alternative Definitions of Vacuity

Let  $\psi$  be a subformula of  $\varphi$ . We give three definitions of when  $\psi$  *does not affect*  $\varphi$ , and compare them. We refer to the definition of [BBER97] as *formula vacuity*. We give two new definitions, *trace vacuity* and *structure vacuity*, according to trace and formula semantics. We are only interested in the cases where  $\varphi$  is satisfied in the structure.

Intuitively,  $\psi$  does not affect  $\varphi$  in  $M$  if we can perturb  $\psi$  without affecting the truth of  $\varphi$  in  $M$ . In previous work, syntactic perturbation was allowed. Using UQLTL we formalize the concept of semantic perturbation. Instead of changing  $\psi$  syntactically, we directly change the set of points in a structure or on a trace in which it holds.

**Definition 3.** Let  $\varphi$  be a formula satisfied in  $M$  and let  $\psi$  be a subformula of  $\varphi$ .

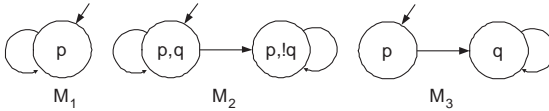
- $\psi$  does not affect<sub>f</sub>  $\varphi$  in  $M$  iff for every LTL formula  $\xi$ ,  $M \models \varphi [\psi \leftarrow \xi]$  [BBER97].
- $\psi$  does not affect<sub>s</sub>  $\varphi$  in  $M$  iff  $M \models_s \forall x \varphi [\psi \leftarrow x]$ .
- $\psi$  does not affect<sub>t</sub>  $\varphi$  in  $M$  iff  $M \models_t \forall x \varphi [\psi \leftarrow x]$ .

We say that  $\psi$  *affects<sub>f</sub>*  $\varphi$  in  $M$  iff it is not the case that  $\psi$  does not affect<sub>f</sub>  $\varphi$  in  $M$ . We say that  $\varphi$  is *formula vacuous* in  $M$ , if there exists a subformula  $\psi$  such that  $\psi$  does not affect<sub>f</sub>  $\varphi$ . We define *affects<sub>s</sub>*, *affects<sub>t</sub>*, *structure vacuity* and *trace vacuity* similarly. Notice that we do not restrict a subformula to occur once and it can be of mixed polarity.

The three semantics form a hierarchy. Structure semantics is the weakest and trace semantics the strongest. Formally, for an LTL formula  $\varphi$  we have the following.

**Lemma 1.** – If  $\psi$  does not affect<sub>t</sub>  $\varphi$  in  $M$ , then  $\psi$  does not affect<sub>f</sub>  $\varphi$  in  $M$  as well.  
 – If  $\psi$  does not affect<sub>f</sub>  $\varphi$  in  $M$ , then  $\psi$  does not affect<sub>s</sub>  $\varphi$  in  $M$  as well.  
 – The reverse implications do not hold.

Which is the most appropriate definition for practical applications? We show that structure and formula vacuity are sensitive to changes in the design that do not relate to the formula. Consider the formula  $\varphi = p \rightarrow Xp$  and models  $M_1$  and  $M_2$  in Figure 1. In  $M_2$  we add a proposition  $q$  whose behavior is independent of  $p$ 's behavior. We would not like formulas that relate to  $p$  to change their truth value or their vacuity. Both  $M_1$  and its extension  $M_2$  satisfy  $\varphi$  and  $\varphi$  relates only to  $p$ . While  $p$  does not affect<sub>f</sub>  $\varphi$  in  $M_1$ , it does affect<sub>f</sub>  $\varphi$  in  $M_2$  (and similarly for affects<sub>s</sub>). Indeed, the formula  $\varphi [p \leftarrow q] = q \rightarrow Xq$  does not hold in  $M_2$ . Note that in both models  $p$  affects<sub>t</sub>  $\varphi$ .



**Fig. 1.** Changes in the design and dependance on syntax.

Formula vacuity is also *sensitive to the specification language*. That is, a formula passing vacuously might pass unvacuously once the specification language is extended.

Consider the Kripke structure  $M_3$  in Figure 1 and the formula  $\varphi = Xq \rightarrow XXq$ . For the single trace  $\pi \in \mathcal{T}(M_3)$ , it holds that  $\pi^2 = \pi^1$ . Thus, every (future) LTL formula is either true along every suffix of  $\pi^1$ , or is false along every such suffix. Hence, subformula  $q$  does not affect<sub>f</sub>  $\varphi$ . We get an opposite result if the specification language is LTL with  $X^{-1}$  [LPZ85]. Formally, for  $\psi$  in LTL,  $M, \pi, i \models X^{-1}(\psi)$  iff  $i > 0$  and  $M, \pi, i - 1 \models \psi$ . Clearly, for every model  $M$  we have  $M, \pi, 0 \not\models X^{-1}(p)$ . In the example,  $M_3 \not\models \varphi [q \leftarrow X^{-1}(p)]$  since  $M_3, \pi, i \models X^{-1}(p)$  iff  $i = 1$ , thus  $q$  affects<sub>f</sub>  $\varphi$ .

To summarize, trace vacuity is preferable since it is not sensitive to changes in the design (unlike structure and formula vacuity) and it is independent of the specification language (unlike formula vacuity). In addition as we show in Section 4, trace vacuity is the only notion of vacuity for which an efficient decision procedure is known to exist.

We claim that if subformulas are restricted to pure polarity, all the definitions of vacuity coincide. In such a case the algorithm proposed in [KV03], to replace the subformula  $\psi$  by  $\perp$  is adequate for vacuity detection according to all three definitions.

**Theorem 2.** *If  $\psi$  is of pure polarity in  $\varphi$  then the following are equivalent.*

1.  $M, \pi, i \models \varphi [\psi \leftarrow \perp]$
2.  $M, \pi, i \models_s \forall x \varphi [\psi \leftarrow x]$
3. *for every formula  $\xi$  we have  $M, \pi, i \models \varphi [\psi \leftarrow \xi]$*
4.  $M, \pi, i \models_t \forall x \varphi [\psi \leftarrow x]$

## 4 Algorithm and Complexity

We give now algorithms for checking vacuity according to the different definitions. We show that the algorithm of [KV03], which replaces a subformula by either **true** or **false** (according to its polarity), cannot be applied to subformulas of mixed polarity. We then study structure and trace vacuity. Decision of formula vacuity remains open.

We show that the algorithm of [KV03] cannot be applied to subformulas of mixed polarity. Consider the Kripke structure  $M_2$  in Figure 1 and the formula  $\varphi = p \rightarrow Xp$ . Clearly,  $M_2 \not\models_s \forall x \varphi [p \leftarrow x]$  (with the structure assignment  $\sigma(x)$  including only the initial state),  $M_2 \not\models \varphi [p \leftarrow q]$ , and  $M_2 \not\models_t \forall x \varphi [p \leftarrow x]$  (with the trace assignment  $\alpha(x) = \{0\}$ ). Hence,  $p$  affects  $\varphi$  according to all three definitions. On the other hand,  $M \models \varphi [p \leftarrow \text{false}]$  and  $M \models \varphi [p \leftarrow \text{true}]$ . We conclude that the algorithm of [KV03] cannot be applied to subformulas of mixed polarity.

We now solve trace vacuity. For a formula  $\varphi$  and a model  $M = \langle AP, S, S_0, R, L \rangle$  where  $M \models \varphi$ , we check whether  $\psi$  affects<sub>t</sub>  $\varphi$  in  $M$  by model checking the UQLTL formula  $\varphi' = \forall x \varphi [\psi \leftarrow x]$  on  $M$ . Subformula  $\psi$  does not affect<sub>t</sub>  $\varphi$  iff  $M \models \varphi'$ . The algorithm in Figure 2 detects if  $\psi$  affects<sub>t</sub>  $\varphi$  in  $M$ . The structure  $M'$  guesses at every step the right assignment for the propositional variable  $x$ . Choosing a path in  $M'$  determines the trace assignment of  $x$ . Formally, we have the following.

**Theorem 3.** [VW94] *Given a structure  $M$  and an LTL formula  $\varphi$ , we can model check  $\varphi$  over  $M$  in time linear in the size of  $M$  and exponential in  $\varphi$  and in space polylogarithmic in the size of  $M$  and quadratic in the length of  $\varphi$ .*

1. Compute the polarity of  $\psi$  in  $\varphi$ .
2. If  $\psi$  is of pure polarity, model check  $M \models \varphi[\psi \leftarrow \perp]$ .
3. Otherwise, construct  $M' = \langle AP \cup \{x\}, S \times 2^x, S_0 \times 2^x, R', L \rangle$ , where for every  $X_1, X_2 \subseteq 2^x$  and  $s_1, s_2 \in S$  we have  $(s_1 \times X_1, s_2 \times X_2) \in R'$  iff  $(s_1, s_2) \in R$ .
4. Model check  $M' \models \varphi[\psi \leftarrow x]$ .

**Fig. 2.** Algorithm for Checking if  $\psi$  Affects<sub>t</sub>  $\varphi$

**Corollary 1.** *Given a structure  $M$  and an LTL formula  $\varphi$  such that  $M \models \varphi$ , we can decide whether subformula  $\psi$  affects<sub>t</sub>  $\varphi$  in time linear in the size of  $M$  and exponential in  $\varphi$  and in space polylogarithmic in the size of  $M$  and quadratic in the length of  $\varphi$ .*

Recall that in symbolic model checking, the modified structure  $M'$  is not twice the size of  $M$  but rather includes just one additional variable. In order to check whether  $\varphi$  is trace vacuous we have to check whether there exists a subformula  $\psi$  of  $\varphi$  such that  $\psi$  does not affect<sub>t</sub>  $\varphi$ . Given a set of subformulas  $\{\psi_1, \dots, \psi_n\}$  we can check whether one of these subformulas does not affect<sub>t</sub>  $\varphi$  by iterating the above algorithm  $n$  times. The number of subformulas of  $\varphi$  is proportional to the size of  $\varphi$ .

**Theorem 4.** *Given a structure  $M$  and an LTL formula  $\varphi$  such that  $M \models \varphi$ . We can check whether  $\varphi$  is trace vacuous in  $M$  in time  $O(|\varphi| \cdot C_M(\varphi))$  where  $C_M(\varphi)$  is the complexity of model checking  $\varphi$  over  $M$ .*

Unlike trace vacuity, there does not exist an efficient algorithm for structure vacuity. We claim that deciding does not affect<sub>s</sub> is co-NP-complete in the structure. Notice, that co-NP-complete in the structure is much worse than PSPACE-complete in the formula. Indeed, the size of the formula is negligible when compared to the size of the model. Co-NP-completeness of structure vacuity renders it completely impractical.

**Lemma 2 (Deciding does not affect<sub>s</sub>).** *For  $\varphi$  in LTL, a subformula  $\psi$  of  $\varphi$  and a structure  $M$ , the problem of deciding whether  $\psi$  does not affect<sub>s</sub>  $\varphi$  in  $M$  is co-NP-complete with respect to the structure  $M$ .*

The complexity of deciding affects<sub>f</sub> is unclear. For subformulas of pure polarity (or occurrences of subformulas) the algorithm of [KV03] is correct. We have found neither a lower bound nor an upper bound for deciding affects<sub>f</sub> in the case of mixed polarity.

## 5 Pragmatic Aspects

**Display of Results.** When applying vacuity detection in an industrial setting there are two options. We can either give the user a simple yes/no answer, or we can accompany a positive answer (vacuity) with a witness formula. Where  $\psi$  does not affect  $\varphi$  we supply  $\varphi[\psi \leftarrow x]$  (or  $\varphi[\psi \leftarrow \perp]$  where  $\psi$  is of pure polarity) as our witness to the vacuity of  $\varphi$ . When we replace a subformula by a constant, we propagate the constants upwards<sup>1</sup>.

<sup>1</sup> I.e. if in subformula  $\theta = \psi_1 \wedge \psi_2$  we replace  $\psi_1$  by **false**, then  $\theta$  becomes **false** and we continue propagating this value above  $\theta$ .

```

active := en  $\wedge$   $\neg$ in ;          rdy_active :=  $\neg$  rdy_out  $\wedge$   $\neg$  active ;
      bsy_active :=  $\neg$  bsy_out  $\wedge$   $\neg$  active;
      active_inactive := rdy_active  $\wedge$   $\neg$  bsy_active ;
two_consecutive :=  $G[(\text{reset} \wedge \text{active\_inactive}) \rightarrow X\neg\text{active\_inactive}]$ ;
two_consecutive[active_inactive2  $\leftarrow \perp$ ] :=  $G\neg(\text{reset} \wedge \text{active\_inactive})$ ;
two_consecutive[en2  $\leftarrow \perp$ ] :=  $G(\text{reset} \wedge \text{active\_inactive}) \rightarrow X\neg(\neg \text{rdy\_out} \wedge \neg \text{bsy\_active})$ ;

```

Fig. 3. Vacuous pass

Previous works suggested that the users of vacuity detection are interested in simple yes / no answers. That is, whether the property is vacuous or not. Suppose that  $\psi$  does not affect  $\varphi$ . It follows that if  $\psi'$  is a subformula of  $\psi$  then  $\psi'$  does not affect  $\varphi$  as well. In order to get a yes / no answer only the minimal subformulas (atomic propositions) of  $\varphi$  have to be checked [BBER97,KV03]. When the goal is to give the user feedback on the source of detected vacuity, it is often more useful to check non-minimal subformulas.

Consider for example the formula *two\_consecutive* in Figure 3. This is an example of a formula that passed vacuously in one of our designs. The reason for the vacuous pass is that one of the signals in *active\_inactive* was set to **false** by a wrong environmental assumption. The formula *two\_consecutive*[*active\_inactive*<sub>2</sub>  $\leftarrow \perp$ ] is the witness to the fact that the second occurrence of *active\_inactive* does not affect *two\_consecutive*. From this witness it is straightforward to understand what is wrong with the formula. The formula *two\_consecutive*[en<sub>2</sub>  $\leftarrow \perp$ ] is the witness associated with the occurrence of the proposition *en* under the second occurrence of *rdy\_active* (after constant propagation). Clearly, this report is much less legible. Thus, it is preferable to check vacuity of non-minimal subformulas and subformula occurrences.

If we consider the formula as represented by a tree (rather than DAG – directed acyclic graph) then the number of leaves (propositions) is proportional to the number of nodes (subformulas). We apply our algorithm from top to bottom. We check whether the maximal subformulas affect the formula. If a subformula does not affect, there is no need to continue checking below it. If a subformula does affect, we continue and check its subformulas. In the worst case, when all the subformulas affect the formula, the number of model checker runs in order to give the most intuitive counter example is double the size of the minimal set (number of propositions). The yes / no view vs. the intuitive witness view offer a clear tradeoff between minimal number of model checker runs (in the worst case) and giving the user the most helpful information. We believe that the user should be given the most comprehensive witness. In our implementation we check whether **all** subformulas and occurrences of subformulas affect the formula.

**Occurrences vs. Subformulas.** We introduced an algorithm that determines if a subformula with multiple occurrences affects a formula. We now give examples in which checking a subformula is more intuitive, and examples in which checking an occurrence is more intuitive. We conclude that a vacuity detection algorithm has to check both.

The following example demonstrates why it is reasonable to check if a subformula affects a formula. Let  $\varphi = G(p \rightarrow p)$ . Intuitively,  $p$  does not affect  $\varphi$ , since every

expression (or variable) implies itself. Indeed, according to all the definitions  $p$  does not affect  $\varphi$ , regardless of the model. However, every occurrence of  $p$  may affect  $\varphi$ . Indeed, both  $Gp = \varphi [p_1 \leftarrow \perp]$  and  $G\neg p = \varphi [p_2 \leftarrow \perp]$  may fail (here,  $p_i$  denotes the  $i$ th occurrence of  $p$ ).

The following example demonstrates why it is reasonable to check if an occurrence affects a formula. Let  $\varphi = p \wedge G(q \rightarrow p)$ . Assume  $q$  is always **false** in model  $M$ . Clearly, the second occurrence of  $p$  does not affect  $\varphi$  in  $M$ . However, the subformula  $p$  does affect  $\varphi$  in  $M$ . Every assignment that gives  $x$  the value **false** at time 0 would falsify the formula  $\varphi [p \leftarrow x]$ . Recall the formula *two\_consecutive* in Figure 3. The vacuous pass in this case is only with respect to occurrences and not to subformulas.

We believe that a *thorough vacuity-detection* algorithm should detect both subformulas and occurrences that do not affect the examined formula. It is up to the user to decide which vacuity alerts to ignore.

**Minimizing the Number of Checks.** We choose to check whether all subformulas and all occurrences of subformulas affect the formula. Applying this policy in practice may result in many runs of the model checker and may be impractical. We now show how to reduce the number of subformulas and occurrences for which we check vacuity by analyzing the structure of the formula syntactically.

As mentioned, once we know that  $\psi$  does not affect  $\varphi$ , there is no point in checking subformulas of  $\psi$ . If  $\psi$  affects  $\varphi$  we have to check also the subformulas of  $\psi$ . We show that in some cases for  $\psi'$  a subformula of  $\psi$  we have  $\psi'$  affects  $\varphi$  iff  $\psi$  affects  $\varphi$ . In these cases there is no need to check direct subformulas of  $\psi$  also when  $\psi$  affects  $\varphi$ .

Suppose the formula  $\varphi$  is satisfied in  $M$ . Consider an occurrence  $\theta_1$  of the subformula  $\theta = \psi_1 \wedge \psi_2$  of  $\varphi$ . We show that if  $\theta_1$  is of positive polarity then  $\psi_i$  affects  $\varphi$  iff  $\theta_1$  affects  $\varphi$  for  $i = 1, 2$ . As mentioned,  $\theta_1$  does not affect  $\varphi$  implies  $\psi_i$  does not affect  $\varphi$  for  $i = 1, 2$ . Suppose  $\theta_1$  affects  $\varphi$ . Then  $M \not\models \varphi [\theta_1 \leftarrow \text{false}]$ . However,  $\varphi [\psi_i \leftarrow \text{false}] = \varphi [\theta_1 \leftarrow \text{false}]$ . It follows that  $M \not\models \varphi [\psi_i \leftarrow \text{false}]$  and that  $\psi_i$  affects  $\varphi$ . In the case that  $\theta_1$  is of negative (or mixed) polarity the above argument is incorrect. Consider the formula  $\varphi = \neg(\psi_1 \wedge \psi_2)$  and a model where  $\psi_1$  never holds. It is straightforward to see that  $\psi_1 \wedge \psi_2$  affects  $\varphi$  while  $\psi_2$  does not affect  $\varphi$ .

It follows that we can analyze  $\varphi$  syntactically and identify occurrences  $\theta_1$  such that  $\theta_1$  affects  $\varphi$  iff the subformulas of  $\theta_1$  affect  $\varphi$ . In these cases, it is sufficient to model check  $\forall x \varphi [\theta_1 \leftarrow x]$ . Below the immediate subformulas of  $\theta_1$  we have to continue with the same analysis. For example, if  $\theta = (\psi_1 \vee \psi_2) \wedge (\psi_3 \wedge \psi_4)$  is of positive polarity and  $\theta$  affects  $\varphi$  we can ignore  $(\psi_1 \vee \psi_2)$ ,  $(\psi_3 \wedge \psi_4)$ ,  $\psi_3$ , and  $\psi_4$ . We do have to check  $\psi_1$  and  $\psi_2$ . In Table 1 we list the operators under which we can apply such elimination. In the polarity column we list the polarities under which the elimination scheme applies to the operator. In the operands column we list the operands that we do not have to check. We stress that below the immediate operands we have to continue applying the analysis.

The analysis that leads to the above table is quite simple. Using a richer set of operators one must use similar reasoning to extend the table. We distinguish between pure and mixed polarity. The above table is true for occurrences. Mixed polarity is only introduced when the specification language includes operators with no polarity (e.g.  $\oplus$ ,  $\leftrightarrow$ ). In order to apply a similar elimination to subformulas with multiple occurrences one has to take into account the polarities of all occurrences and the operator under which

**Table 1.** Operators for which checks can be avoided

Operator	Polarity	Operands
$\wedge$	+	all
$\vee$	-	all
$\neg$	pure / mixed	all

Operator	Polarity	Operands
$X$	pure / mixed	all
$U$	pure	second
$G$	pure	all
$F$	pure	all

every occurrence appears. However, suppose that the subformula  $\theta = f(\psi_1, \psi_2)$  occurs more than once but  $\psi_1$  and  $\psi_2$  occur only under  $\theta$ . In this case, once we check whether  $\theta$  affects  $\varphi$ , the elimination scheme can be applied to  $\psi_1$  and  $\psi_2$ .

**Implementation and Methodology.** We implemented the above algorithms in one of Intel’s formal verification environments. We use the language ForSpec [AFF<sup>+</sup>02] with the BDD-based model checker Forecast [FKZ<sup>+</sup>00] and the SAT-based bounded model checker Thunder [CFF<sup>+</sup>01]. The users can decide whether they want thorough vacuity detection or just to specify which subformulas / occurrences to check. In the case of thorough vacuity detection, for every subformula and every occurrence (according to the elimination scheme above) we create one witness formula. The vacuity algorithm amounts to model checking each of the witnesses. Both model checkers are equipped with a mechanism that allows model checking of many properties simultaneously.

The current methodology of using vacuity is applying thorough vacuity on every specification. The users prove that the property holds in the model; then, vacuity of the formula is checked. If applying thorough vacuity is not possible (due to capacity problems), the users try to identify the important subformulas and check these subformulas manually. In our experience, vacuity checks proved to be effective mostly when the pruning and assumptions used in order to enable model checking removed some important part of the model, thus rendering the specification vacuously true. In many examples vacuity checking pointed out to such problems. We also have cases where vacuity pointed out redundant parts in the specification.

In Table 2 we include some experimental results. We used real-life examples from processor designs. We include these results in order to give the flavor of the performance of vacuity checking. Each line in the table corresponds to one property. Some properties are the conjunction of a few assertions. In such a case, every assertion is checked separately (both in model checking and vacuity detection). For each property we report on 4 different experiments. The first column specifies the number of witness formulas sent to the model checker for vacuity detection. The number in parentheses indicates the number of non-affecting subformulas / occurrences. The block titled Forecast reports on three experiments. The column titled *MC* reports on the results of model checking the property itself. The column titled *Vacuity* reports on the results of model checking all the witness formulas for all the assertions. Finally, the column titled *Combined* reports on the results of model checking all the witnesses with all the assertions. In each column we specify the time (in seconds) and space (BDD nodes) required by Forecast. The symbol ! indicates that Forecast timed out. Recall that in vacuity detection we hope that all the witness formulas do not pass in the model. As bounded model checking is especially

adequate for falsification, we prove the correctness of the property using Forecast and falsify the witness formulas using Thunder. Witness formulas that Thunder was unable to falsify can be checked manually using Forecast. The last column reports on the results of model checking all the witness formulas for all the assertions using Thunder. We write the time (in seconds) required by Thunder, and ! in case that Thunder did not terminate in 8 hours. In the case that Thunder did not terminate we report (in brackets) the bound up to which the formulas were checked<sup>2</sup>. We ran the examples on a Intel(R) Pentium™ 4 2.2GHz processor running Linux with 2GByte memory. Notice that some of these examples pass vacuously.

**Table 2.** Experimental results

Property	# Checks	Forecast						Thunder
		MC		Vacuity		Combined		
		Time	Nodes	Time	Nodes	Time	Nodes	
check_internal_sig	5(1)	1936	3910K	2051	2679K	3185	5858K	2.28(0)
lsd_indication	17(5)	1699	2150K	2265	2566K	1986	3483K	!(5)[40]
directive	4(0)	611	1120K	16132	4945K	7355	8943K	25(0)
forbidden_start	2(0)	532	549K	1859	4064K	2422	4274K	22(0)
nested_start	22 (13)	737	1294K	11017	6153K	10942	6153K	!(18)[70]
pilot_session	129(? <sup>3</sup> )	5429	3895K	67126!	25366K	66157!	20586K	!(16)[60]
new_code	31(1)	1265	2455K	1765	2853K	3097	3932	!(1)[50]

## 6 Summary and Future Work

We investigated vacuity detection with respect to subformulas with multiple occurrences. We re-examined the basic intuition underlying the concept of vacuity, which until now has been defined as sensitivity with respect to syntactic perturbation. We studied sensitivity with respect to semantic perturbation, which we modeled by universal propositional quantification. We showed that this yields a hierarchy of vacuity notions. We argued that the right notion is that of vacuity defined with respect to traces, described an algorithm for vacuity detection, and discussed pragmatic aspects.

We were motivated by the need to extend vacuity detection to industrial-strength property-specification languages such as ForSpec [AFF<sup>+</sup>02]. ForSpec is significantly richer syntactically and semantically than LTL. Our vacuity-detection algorithm for subformulas of mixed polarity can handle ForSpec's rich set of arithmetical and Boolean operators. ForSpec's semantic richness is the result of its *regular layer*, which includes regular events and formulas constructed from regular events. The extension of vacuity detection to ForSpec's regular layer will be described in a future paper.

<sup>2</sup> Note that in the case of *lsd\_indication* and *new\_code* the partial answer is in fact the final answer, as can be seen from the run of Forecast

<sup>3</sup> For this property, we do not know the number of non affecting subformulas / occurrences. There are 16 non affecting subformulas / occurrences up to bound 60.

## 7 Acknowledgments

We thank R. Fraer, D. Lichten, and K. Sajid for their contribution and useful remarks.

## References

- [AFF<sup>+</sup>02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification language. In *8th TACAS*, LNCS 2280, 296–311, 2002. Springer.
- [BB94] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *31st DAC*, IEEE Computer Society, 1994.
- [BBER97] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *9th CAV*, LNCS 1254, 279–290, 1997. Full version in *FMSD*, 18 (2): 141–162, 2001.
- [CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *WLP*, LNCS 131, 52–71. 1981.
- [CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2):244–263, 1986.
- [CFF<sup>+</sup>01] F. Cofy, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M.Y. Vardi. Benefits of bounded model checking at an industrial setting. In *13th CAV*, 2001.
- [CGMZ95] E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd DAC*, 1995.
- [CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [Eme90] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, chapter 16. Elsevier, MIT press, 1990.
- [FKZ<sup>+</sup>00] R. Fraer, G. Kamhi, B. Ziv, M. Vardi, and L. Fix. Prioritized traversal: efficient reachability analysis for verification and falsification. In *12th CAV*, LNCS 1855, 2000.
- [Kup95] O. Kupferman. Augmenting branching temporal logics with existential quantification over atomic propositions. In *8th CAV*, LNCS 939, 1995.
- [Kur98] R.P. Kurshan. *FormalCheck User's Manual*. Cadence Design, Inc., 1998.
- [KV99] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. In *10th CHARME*, LNCS 170, 82–96. Springer-Verlag, 1999.
- [KV03] O. Kupferman and M.Y. Vardi. Vacuity detection in temporal model checking. *STTT*, 4(2):224–233, 2003.
- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *12th POPL*, 97–107, 1985.
- [LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, LNCS 193, 196–218, 1985. Springer-Verlag.
- [Pnu77] A. Pnueli. The temporal logic of programs. In *18th FOCS*, 46–57, 1977.
- [PP95] B. Plessier and C. Pixley. Formal verification of a commercial serial bus interface. In *14th IEEE Conf. on Computers and Comm.*, 378–382, 1995.
- [PS02] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *14th CAV*, LNCS 2404, 485–499. Springer-Verlag, 2002.
- [QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, LNCS 137, 1981.
- [SVW85] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. In *10th ICALP*, LNCS 194, 1985.
- [VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st LICS*, 332–344, 1986.
- [VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *IC*, 115(1), 1994.



# Bridging the Gap between Fair Simulation and Trace Inclusion<sup>\*</sup>

Yonit Kesten<sup>1</sup>, Nir Piterman<sup>2</sup>, and Amir Pnueli<sup>2</sup>

<sup>1</sup> Ben Gurion University, Beer-Sheva, Israel. ykesten@bgumail.bgu.ac.il

<sup>2</sup> Weizmann Institute, Rehovot, Israel. (nirp, amir)@wisdom.weizmann.ac.il

**Abstract.** The paper considers the problem of checking abstraction between two finite-state *fair discrete systems*. In automata-theoretic terms this is trace inclusion between two Streett automata. We propose to reduce this problem to an algorithm for checking fair simulation between two generalized Büchi automata. For solving this question we present a new triply nested  $\mu$ -calculus formula which can be implemented by symbolic methods.

We then show that every trace inclusion of this type can be solved by fair simulation, provided we augment the concrete system (the contained automaton) by appropriate auxiliary variables. This establishes that fair simulation offers a complete method for checking trace inclusion. We illustrate the feasibility of the approach by algorithmically checking abstraction between finite state systems whose abstraction could only be verified by deductive methods up to now.

## 1 Introduction

A frequently occurring problem in verification of reactive systems is the problem of *abstraction* (symmetrically *refinement*) in which we are given a concrete reactive system  $C$  and an abstract reactive system  $A$  and are asked to check whether  $A$  *abstracts*  $C$ , denoted  $C \sqsubseteq A$ . In the linear-semantics framework this question calls for checking whether any observation of  $C$  is also an observation of  $A$ . For the case that both  $C$  and  $A$  are finite-state systems with weak and strong fairness this problem can be reduced to the problem of language inclusion between two Streett automata (e.g., [Var91]).

In theory, this problem has an exponential-time algorithmic solution based on the complementation of the automaton representing the abstract system  $A$  [Saf92]. However, the complexity of this algorithm makes its application prohibitively expensive. For example, our own interest in the finite-state abstraction problem stems from applications of the verification method of *network invariants* [KP00a, KPSZ02, WL89]. In a typical application of this method, we are asked to verify the abstraction  $P_1 \parallel P_2 \parallel P_3 \parallel P_4 \sqsubseteq P_5 \parallel P_6 \parallel P_7$ , claiming that 3 parallel copies of the dining philosophers process abstract 4 parallel copies of the same process. The system on the right has about 1800 states. Obviously, to complement a Streett automaton of 1800 states is hopelessly expensive.

A partial but more effective solution to the problem of checking abstraction between systems (trace inclusion between automata) is provided by the notion of *simulation*.

---

<sup>\*</sup> This research was supported in part by THE ISRAEL SCIENCE FOUNDATION (grant no.106/02-1) and the John von-Neumann Minerva center for Verification of Reactive Systems.

Introduced first by Milner [Mil71], we say that system  $A$  simulates system  $C$ , denoted  $C \preceq A$ , if there exists a *simulation relation*  $R$  between the states of  $C$  and the states of  $A$ . It is required that if  $(s_C, s_A) \in R$  and system  $C$  can move from state  $s_C$  to state  $s'_C$ , then system  $A$  can move from  $s_A$  to some  $s'_A$  such that  $(s'_C, s'_A) \in R$ . Additional requirements on  $R$  are that if  $(s_C, s_A) \in R$  then  $s_C$  and  $s_A$  agree on the values of their observables, and for every  $s_C$  initial in  $C$  there exists  $s_A$  initial in  $A$  such that  $(s_C, s_A) \in R$ . It is obvious that  $C \preceq A$  is a sufficient condition for  $C \sqsubseteq A$ . For finite-state systems, we can check  $C \preceq A$  in time proportional to  $(|\Sigma_C| \cdot |\Sigma_A|)^2$  where  $\Sigma_C$  and  $\Sigma_A$  are the sets of states of  $A$  and  $C$  respectively [BR96, HHK95].

While being a sufficient condition, simulation is definitely not a necessary condition for abstraction. This is illustrated by the two systems presented in Fig. 1

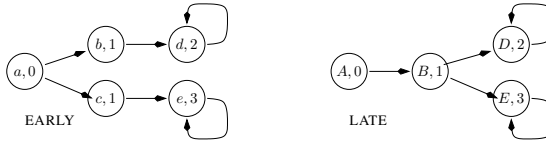


Fig. 1. Systems EARLY and LATE

The labels in these two systems consist of a local state name (a–e, A–E) and an observable value. Clearly these two systems are (observation)-equivalent because they each have the two possible observations  $012^\omega + 013^\omega$ . Thus, each of them abstracts the other. However, when we examine their simulation relation, we find that  $\text{EARLY} \preceq \text{LATE}$  but  $\text{LATE} \not\preceq \text{EARLY}$ . This example illustrates that, in some cases we can use simulation in order to establish abstraction (trace inclusion) but this method is not complete.

The above discussion only covered the case that  $C$  and  $A$  did not have any fairness constraints. There were many suggestions about how to enhance the notion of simulation in order to account for fairness [GL94, LT87, HKR97, HR00]. The one we found most useful for our purposes is the definition of fair simulation from [HKR97]. Henzinger et al. proposed a game-based view of simulation. As in the unfair case, the definition assumes an underlying simulation relation  $R$  which implies equality of the observables. However, in the presence of fairness, it is not sufficient to guarantee that every step of the concrete system can be matched by an abstract step with corresponding observables. Here we require that the abstract system has a *strategy* such that any joint run of the two systems, where the abstract player follows this strategy either satisfies the fairness requirements of the abstract system or fails to satisfy the fairness requirements of the concrete system. This guarantees that every concrete observation has a corresponding abstract observation with matching values of the observables.

**Algorithmic Considerations.** In order to determine whether one system fairly simulates another (*solve fair simulation*) we have to solve games [HKR97]. When the two systems in question are reactive systems with strong fairness (Streett), the winning condition of the resulting game is an implication between two Streett conditions (*fsim-games*). In [HKR97] the solution of fsim-games is reduced to the solution of Streett games. In [KV98] an algorithm for solving Streett games is presented. The time complexity of this

approach is  $(|\Sigma_A| \cdot |\Sigma_C| \cdot (3^{k_A} + k_C))^{2k_A + k_C} \cdot (2k_A + k_C)!$  where  $k_C$  and  $k_A$  denote the number of Streett pairs of  $C$  and  $A$ . Clearly, this complexity is too high. It is also not clear whether this algorithm can be implemented symbolically.

In the context of fair simulation, Streett systems cannot be reduced to simpler systems [KPV00]. That is, in order to solve the question of fair simulation between Streett systems we have to solve fsm-games in their full generality. However, we are only interested in fair simulation as a precondition for trace inclusion. In the context of trace inclusion we can reduce the problem of two reactive systems with strong fairness to an equivalent problem with weak fairness. Formally, for the reactive systems  $C$  and  $A$  with Streett fairness requirements, we construct  $C^B$  and  $A^B$  with generalized Büchi requirements, such that  $C \sqsubseteq A$  iff  $C^B \sqsubseteq A^B$ . Solving fair simulation between  $C^B$  and  $A^B$  is simpler. The winning condition of the resulting game is an implication between two generalized Büchi conditions (denoted *generalized Streett*[1]).

In [dAHM01], a solution for games with winning condition expressed as a general LTL formula is presented. The algorithm in [dAHM01] constructs a deterministic parity word automaton for the winning condition. The automaton is then converted into a  $\mu$ -calculus formula that evaluates the set of winning states for the relevant player.

In [EL86], Emerson and Lei show that a  $\mu$ -calculus formula is in fact a recipe for symbolic model checking<sup>1</sup>. The main factor in the complexity of  $\mu$ -calculus model checking is the *alternation depth* of the formula. The symbolic algorithm for model checking a  $\mu$ -calculus formula of alternation depth  $k$  takes time proportional to  $(mn)^k$  where  $m$  is the size of the formula and  $n$  is the size of the model [EL86].

In fsm-games the winning condition is an implication between two Streett conditions. A deterministic Streett automaton for this winning condition has  $3^{k_A} \cdot k_C$  states and  $2k_A + k_C$  pairs. A deterministic parity automaton for the same condition has  $3^{k_A} \cdot k_C \cdot (2k_A + k_C)!$  states and index  $4k_A + 2k_C$ . The  $\mu$ -calculus formula constructed by [dAHM01] is of alternation depth  $4k_A + 2k_C$  and proportional in size to  $3^{k_A} \cdot k_C \cdot (2k_A + k_C)!$ . Hence, in this case, there is no advantage in using [dAHM01].

In the case of generalized Streett[1] games, a deterministic parity automaton for the winning condition has  $|J_C| \cdot |J_A|$  states and index 3, where  $|J_C|$  and  $|J_A|$  denote the number of Büchi sets in the fairness of  $C^B$  and  $A^B$  respectively. The  $\mu$ -calculus formula of [dAHM01] is proportional to  $3|J_C| \cdot |J_A|$  and has alternation depth 3.

We give an alternative  $\mu$ -calculus formula that solves generalized Streett[1] games. Our formula is also of alternation depth 3 but its length is proportional to  $2|J_C| \cdot |J_A|$  and it is simpler than that of [dAHM01]. Obviously, our algorithm is tailored for the case of generalized-Streett[1] games while [dAHM01] give a generic solution for any LTL game<sup>2</sup>. The time complexity of solving fair simulation between two reactive systems after converting them to systems with generalized Büchi fairness requirements is  $(|\Sigma_A| \cdot |\Sigma_C| \cdot 2^{k_A + k_C} \cdot (|J_A| + |J_C| + k_A + k_C))^3$ .

<sup>1</sup> There are more efficient algorithms for  $\mu$ -calculus model checking [Jur00]. However, Jurdzinski's algorithm cannot be implemented symbolically.

<sup>2</sup> One may ask why not take one step further and convert the original reactive systems to Büchi systems. In this case, the induced game is a parity[3] game and there is a simple algorithm for solving it. Although both algorithms work in cubic time, the latter performed much worse than the one described above.

**Making the Method Complete.** Even if we succeed to present a complexity-acceptable algorithm for checking fair simulation between generalized-Büchi systems, there is still a major drawback to this approach which is its incompleteness. As shown by the example of Fig. 1, there are (trivially simple) systems  $C$  and  $A$  such that  $C \sqsubseteq A$  but this abstraction cannot be proven using fair simulation. Fortunately, we are not the first to be concerned by the incompleteness of simulation as a method for proving abstraction. In the context of infinite-state system verification, Abadi and Lamport studied the method of simulation using an *abstraction mapping* [AL91]. It is not difficult to see that this notion of simulation is the infinite-state counterpart of the fair simulation as defined in [HKR97] but restricted to the use of memory-less strategies. However, [AL91] did not stop there but proceeded to show that if we are allowed to add to the concrete system auxiliary *history* and *prophecy* variables, then the simulation method becomes *complete*. That is, with appropriate augmentation by auxiliary variables, every abstraction relation can be proven using fair simulation. History variables remove the restriction to memory-less strategies, while prophecy variables allow to predict the future and use fair simulation to establish, for example, the abstraction  $\text{LATE} \sqsubseteq \text{EARLY}$ .

The application of Abadi-Lamport, being deductive in nature, requires the users to decide on the appropriate history and prophecy variables, and then design their abstraction mapping which makes use of these auxiliary variables. Implementing these ideas in the finite-state (and therefore algorithmic) world, we expect the strategy (corresponding to the abstraction mapping) to be computed fully automatically. Thus, in our implementation, the user is still expected to identify the necessary auxiliary history or prophecy variables, but following that, the rest of the process is automatic. For example, wishing to apply our algorithm in order to check the abstraction  $\text{LATE} \sqsubseteq \text{EARLY}$ , the user has to specify the augmentation of the concrete system by a temporal tester for the LTL formula  $\Diamond(x = 2)$ . Using this augmentation, the algorithm manages to prove that the augmented system ( $\text{LATE} + \text{tester}$ ) is fairly simulated (hence abstracted) by  $\text{EARLY}$ .

In summary, the contributions of this paper are:

1. Suggesting the usage of fair simulation as a precondition for abstraction between two reactive systems (Streett automata).
2. Observing that in the context of fair simulation for checking abstraction we can simplify the game acceptance condition from implication between two Streett conditions to implication between two generalized Büchi conditions.
3. Providing a more efficient  $\mu$ -calculus formula and its implementation by symbolic model-checking tools for solving the fair simulation between two generalized Büchi systems.
4. Claiming and demonstrating the completeness of the fair-simulation method for proving abstraction between two systems, at the price of augmenting the concrete system by appropriately chosen “observers” and “testers”.

## 2 The Computational Model

As a computational model, we take the model of *fair discrete system* (FDS) [KP00b]. An FDS  $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$  consists of the following components.

- $V = \{u_1, \dots, u_n\}$  : A finite set of typed *state variables* over finite domains. A *state*  $s$  is a type-consistent interpretation of  $V$ . We denote by  $\Sigma$  the set of all states.
- $\mathcal{O} \subseteq V$  : A subset of externally *observable variables*.
- $\Theta$  : The *initial condition*. An assertion characterizing all the initial states.
- $\rho$  : A *transition relation*. This is an assertion  $\rho(V, V')$ , relating a state  $s \in \Sigma$  to its  $\mathcal{D}$ -successor  $s' \in \Sigma$  by referring to both unprimed and primed versions of the state variables. State  $s'$  is a  $\mathcal{D}$ -successor of  $s$  if  $\langle s, s' \rangle \models \rho(V, V')$ , where  $\langle s, s' \rangle$  is the joint interpretation which interprets  $x \in V$  as  $s[x]$ , and  $x'$  as  $s'[x]$ .
- $\mathcal{J} = \{J_1, \dots, J_k\}$  : Assertions expressing the *justice (weak fairness)* requirements.
- $\mathcal{C} = \{\langle p_1, q_1 \rangle, \dots, \langle p_n, q_n \rangle\}$  : Assertions expressing the *compassion (strong fairness)* requirements.

We require that every state  $s \in \Sigma$  has at least one  $\mathcal{D}$ -successor. This is ensured by including in  $\rho$  the *idling* disjunct  $V' = V$ . Let  $\sigma : s_0, s_1, \dots$ , be a sequence of states,  $\varphi$  be an assertion, and  $j \geq 0$  be a natural number. We say that  $j$  is a  $\varphi$ -position of  $\sigma$  if  $s_j$  is a  $\varphi$ -state. Let  $\mathcal{D}$  be an FDS for which the above components have been identified. A *run* of  $\mathcal{D}$  is an infinite sequence of states  $\sigma : s_0, s_1, \dots$ , satisfying following:

- *Initiality*:  $s_0$  is initial, i.e.,  $s_0 \models \Theta$ .
- *Consecution*: For  $j = 0, 1, \dots$ , the state  $s_{j+1}$  is a  $\mathcal{D}$ -successor of the state  $s_j$ .

A run of  $\mathcal{D}$  is called a *computation* if it satisfies the following:

- *Justice*: For each  $J \in \mathcal{J}$ ,  $\sigma$  contains infinitely many  $J$ -positions
- *Compassion*: For each  $\langle p, q \rangle \in \mathcal{C}$ , if  $\sigma$  contains infinitely many  $p$ -positions, it must also contain infinitely many  $q$ -positions.

Let  $\text{runs}(\mathcal{D})$  denote the set of runs of  $\mathcal{D}$  and  $\text{Comp}(\mathcal{D})$  the set of computations of  $\mathcal{D}$ .

Systems  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are *compatible* if the intersection of their variables is observable in both systems. For compatible systems  $\mathcal{D}_1$  and  $\mathcal{D}_2$ , we define their *asynchronous parallel composition*, denoted by  $\mathcal{D}_1 \parallel \mathcal{D}_2$ , and the *synchronous parallel composition*, denoted by  $\mathcal{D}_1 \parallel\!\!\parallel \mathcal{D}_2$ , in the usual way [KP00a]. The primary use of synchronous composition is for combining a system with a *tester*  $T_\varphi$  for an LTL formula  $\varphi$ .

The *observations* of  $\mathcal{D}$  are the projection  $\mathcal{D} \downarrow_{\mathcal{O}}$  of  $\mathcal{D}$ -computations onto  $\mathcal{O}$ . We denote by  $\text{Obs}(\mathcal{D})$  the set of all observations of  $\mathcal{D}$ . Systems  $\mathcal{D}_C$  and  $\mathcal{D}_A$  are said to be *comparable* if there is a one to one correspondence between their observable variables. System  $\mathcal{D}_A$  is said to be an *abstraction* of the comparable system  $\mathcal{D}_C$ , denoted  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ , if  $\text{Obs}(\mathcal{D}_C) \subseteq \text{Obs}(\mathcal{D}_A)$ . The abstraction relation is reflexive and transitive. It is also *property restricting*. That is, if  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$  then  $\mathcal{D}_A \models p$  implies that  $\mathcal{D}_C \models p$  for an LTL property  $p$ . We say that two comparable FDS's  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are *equivalent*, denoted  $\mathcal{D}_1 \sim \mathcal{D}_2$  if  $\text{Obs}(\mathcal{D}_1) = \text{Obs}(\mathcal{D}_2)$ . For compatibility with automata terminology, we refer to the observations of  $\mathcal{D}$  also as the *traces* of  $\mathcal{D}$ .

All our concrete examples are given in SPL, which is used to represent concurrent programs (e.g., [MP95, MAB<sup>+</sup>94]). Every SPL program can be compiled into an FDS in a straightforward manner.

**From FDS to JDS.** An FDS with no compassion requirements is called a *just discrete system* (JDS). Let  $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$  be an FDS such that  $\mathcal{C} = \{(p_1, q_1), \dots, (p_m, q_m)\}$  and  $m > 0$ . We define a JDS  $\mathcal{D}^B : \langle V^B, \mathcal{O}^B, \Theta^B, \rho^B, \mathcal{J}^B, \emptyset \rangle$  equivalent to  $\mathcal{D}$ , as follows:

- $V^B = V \cup \{n\_p_i : \mathbf{Boolean} \mid (p_i, q_i) \in \mathcal{C}\} \cup \{x_c\}$ .
- $\mathcal{O}^B = \mathcal{O}$ .
- $\Theta^B = \Theta \wedge x_c = 0 \wedge \bigwedge_{(p_i, q_i) \in \mathcal{C}} n\_p_i = 0$ .
- $\rho^B = \rho \wedge \rho_{n\_p} \wedge \rho_c$ , where

$$\rho_{n\_p} : \bigwedge_{(p_i, q_i) \in \mathcal{C}} (n\_p_i \rightarrow n\_p'_i) \quad \rho_c : x'_c = (x_c \vee \bigvee_{(p_i, q_i) \in \mathcal{C}} (p_i \wedge n\_p_i))$$

- $\mathcal{J}^B = \mathcal{J} \cup \{\neg x_c\} \cup \{n\_p_i \vee q_i \mid (p_i, q_i) \in \mathcal{C}\}$ .

The transformation of an FDS to a JDS follows the transformation of Streett automata to generalized Büchi Automata (see [Cho74] for finite state automata and [Var91] for infinite state automata). We add one Boolean variable  $n\_p_i$  per compassion requirement. This variable is initialized to 0, it can be set nondeterministically to 1 and is never reset. The nondeterministic setting is in fact a guess that no more  $p_i$  states are encountered. Accordingly, we add the justice  $n\_p_i \vee q_i$  so either  $n\_p_i$  is set (and  $p_i$  is false from that point) or  $q_i$  is visited infinitely often. We add one additional variable  $x_c$  initialized to 0, set to 1 at a point satisfying  $\bigvee_{i=1}^m (p_i \wedge n\_p_i)$  and never reset. Once  $x_c$  is set it indicates a *mis-prediction*. We guessed wrong that some  $p_i$  never holds anymore. We add the justice requirement  $\neg x_c$  to ensure that a run in which  $x_c$  is set, is not a computation.

### 3 Simulation Games

Let  $\mathcal{D}_C : \langle V_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$  and  $\mathcal{D}_A : \langle V_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$  be two comparable FDS's. We denote by  $\Sigma_C$  and  $\Sigma_A$  the sets of states of  $\mathcal{D}_C$  and  $\mathcal{D}_A$  respectively. We define the *simulation game structure* (SGS) associated with  $\mathcal{D}_C$  and  $\mathcal{D}_A$  to be the tuple  $G : \langle \mathcal{D}_C, \mathcal{D}_A \rangle$ . A *state* of  $G$  is a type-consistent interpretation of the variables in  $V_C \cup V_A$ . We denote by  $\Sigma_G$  the set of states of  $G$ . We say that a state  $s \in \Sigma_G$  is *correlated*, if  $s[\mathcal{O}_C] = s[\mathcal{O}_A]$ . We denote by  $\Sigma_{cor} \subset \Sigma_G$  the subset of correlated states.

For two states  $s$  and  $t$  of  $G$ ,  $t$  is an *A-successor* of  $s$  if  $(s, t) \models \rho_A$  and  $s[V_C] = t[V_C]$ . Similarly,  $t$  is a *C-successor* of  $s$  if  $(s, t) \models \rho_C$  and  $s[V_A] = t[V_A]$ . A *run* of  $G$  is a maximal sequence of states  $\sigma : s_0, s_1, \dots$  satisfying the following:

- *Consecution*: For each  $j = 0, \dots$ ,  $\blacksquare s_{2j+1}$  is a *C-successor* of  $s_{2j}$ .  
 $\blacksquare s_{2j+2}$  is a *A-successor* of  $s_{2j+1}$ .
- *Correlation*: For each  $j = 0, \dots$ ,  $\blacksquare s_{2j} \in \Sigma_{cor}$

We say that a run is *initialized* if it satisfies

- *Initiality*:  $s_0 \models \Theta_A \wedge \Theta_C$

Let  $G$  be an SGS and  $\sigma$  be a run of  $G$ . The run  $\sigma$  can be viewed as a two player game. Player  $C$ , represented by  $\mathcal{D}_C$ , taking  $\rho_C$  transitions from even numbered states and player  $A$ , represented by  $\mathcal{D}_A$ , taking  $\rho_A$  transitions from odd numbered states. The observations of the two players are correlated on all even numbered states of a run.

A run  $\sigma$  is *winning for player A* if it is infinite and either  $\sigma \Downarrow_{V_C}$  is not a computation of  $\mathcal{D}_C$  or  $\sigma \Downarrow_{V_A}$  is a computation of  $\mathcal{D}_A$ , i.e. if  $\sigma \models \mathcal{F}_C \rightarrow \mathcal{F}_A$ , where for  $\eta \in \{A, C\}$ ,

$$\mathcal{F}_\eta : \bigwedge_{J \in \mathcal{J}_\eta} \square \Diamond J \wedge \bigwedge_{(p, q) \in \mathcal{C}_\eta} (\square \Diamond p \rightarrow \square \Diamond q).$$

Otherwise,  $\sigma$  is *winning for player C*.

Let  $D_A$  and  $D_C$  be some finite domains, intended to record facts about the past history of a computation (serve as a memory). A *strategy* for player  $A$  is a partial function  $f_A : D_A \times \Sigma_G \mapsto D_A \times \Sigma_{cor}$  such that if  $f_A(d, s) = (d', s')$  then  $s'$  is an  $A$ -successor of  $s$ . A *strategy* for player  $C$  is a partial function  $f_C : D_C \times \Sigma_{cor} \mapsto D_C \times \Sigma_G$  such that if  $f_C(d, s) = (d', s')$  then  $s'$  is a  $C$ -successor of  $s$ . Let  $f_A$  be a strategy for player  $A$ , and  $s_0 \in \Sigma_{cor}$ . A run  $s_0, s_1, \dots$  is said to be *compliant* with strategy  $f_A$  if there exists a sequence of  $D_A$ -values  $d_0, d_2, \dots, d_{2j}, \dots$  such that  $(d_{2j+2}, s_{2j+2}) = f_A(d_{2j}, s_{2j+1})$  for every  $j \geq 0$ . Strategy  $f_A$  is *winning* for player  $A$  from state  $s \in \Sigma_{cor}$  if all  $s$ -runs (runs departing from  $s$ ) which are compliant with  $f_A$  are winning for  $A$ . A winning strategy for player  $C$  is defined similarly. We denote by  $W_A$  the set of states from which there exists a winning strategy for player  $A$ . The set  $W_C$  is defined similarly.

An SGS  $G$  is called *determinate* if the sets  $W_A$  and  $W_C$  define a partition on  $\Sigma_{cor}$ . It is well known that every SGS is determinate [GH82].

### 3.1 $\mu$ -Calculus

We define  $\mu$ -calculus [Koz83] over game structures. Consider two FDS's  $\mathcal{D}_C : \langle V_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$ ,  $\mathcal{D}_A : \langle V_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$  and the SGS  $G : \langle \mathcal{D}_C, \mathcal{D}_A \rangle$ . For every variable  $v \in V_C \cup V_A$  the formula  $v = i$  where  $i$  is a constant that is type consistent with  $v$  is an *atomic formula* (p). Let  $V = \{X, Y, \dots\}$  be a set of *relational variables*. Each relational variable can be assigned a subset of  $\Sigma_{cor}$ . The  $\mu$ -calculus formulas are constructed as follows.

$$f ::= p \mid \neg p \mid X \mid f \vee f \mid f \wedge f \mid \otimes f \mid \ominus f \mid \mu X f \mid \nu X f$$

A formula  $f$  is interpreted as the set of states in  $\Sigma_{cor}$  in which  $f$  is true. We write such set of states as  $[[f]]_G^e$  where  $G$  is the SGS and  $e : V \rightarrow 2^{\Sigma_{cor}}$  is an *environment*. The set  $[[f]]_G^e$  is defined for the operators  $\otimes$  and  $\ominus$  as follows.

- $[[\otimes f]]_G^e = \{s \in \Sigma_{cor} \mid \forall t, (s, t) \models \rho_C \rightarrow \exists s', (t, s') \models \rho_A \text{ and } s' \in [[f]]_G^e\}.$
- $[[\ominus f]]_G^e = \{s \in \Sigma_{cor} \mid \exists t, (s, t) \models \rho_C \text{ and } \forall s', (t, s') \models \rho_A \rightarrow s' \in [[f]]_G^e\}.$

For the rest of the operators the semantics is as in the usual definition [Eme97]. The *alternation depth* of a formula is the number of alternations in the nesting of least and greatest fixpoints. A  $\mu$ -calculus formula defines a symbolic algorithm for computing  $[[f]]$  [EL86]. For a  $\mu$ -calculus formula of alternation depth  $k$ , the run time of this algorithm is  $|\Sigma_{cor}|^k$ . For a full exposition of  $\mu$ -calculus we refer the reader to [Eme97].

## 4 Trace Inclusion and Fair Simulation

In the following, we summarize our solution to verifying abstraction between two FDS's systems, or equivalently, trace inclusion between two Streett automata.

Let  $\mathcal{D}_C : \langle V_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$  and  $\mathcal{D}_A : \langle V_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$  be two comparable FDS's. We want to verify that  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ . The best algorithm for solving abstraction is exponential [Saf92]. We therefore advocate to verify fair simulation

[HKR97] as a precondition for abstraction. We adopt the definition of fair simulation presented in [HKR97]. Given  $\mathcal{D}_C$  and  $\mathcal{D}_A$ , we form the SGS  $G : \langle \mathcal{D}_C, \mathcal{D}_A \rangle$ . We say that  $S \subseteq \Sigma_{cor}$  is a *fair-simulation* between  $\mathcal{D}_A$  and  $\mathcal{D}_C$  if there exists a strategy  $f_A$  such that every  $f_A$ -compliant run  $\sigma$  from a state  $s \in S$  is winning for player  $A$  and every even state in  $\sigma$  is in  $S$ . We say that  $\mathcal{D}_A$  *fairly-simulates*  $\mathcal{D}_C$ , denoted  $\mathcal{D}_C \preceq_f \mathcal{D}_A$ , if there exists a fair-simulation  $S$  such that for every state  $s_C \in \Sigma_C$  satisfying  $s_C \models \Theta_C$  there exists a state  $t \in S$  such that  $t \Downarrow_{V_C} = s_C$  and  $t \models \Theta_A$ .

*Claim.* [HKR97] If  $\mathcal{D}_C \preceq_f \mathcal{D}_A$  then  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ . The reverse implication does not hold.

It is shown in [HKR97] that we can determine whether  $\mathcal{D}_C \preceq_f \mathcal{D}_A$  by computing the set  $W_A \subseteq \Sigma_{cor}$  of states which are winning for  $A$  in the SGS  $G$ . If for every state  $s_C \in \Sigma_C$  satisfying  $s_C \models \Theta_C$  there exists some state  $t \in W_A$  such that  $t \Downarrow_{V_C} = s_C$  and  $t \models \Theta_A$ , then  $\mathcal{D}_C \preceq_f \mathcal{D}_A$ .

Let  $k_C = |\mathcal{C}_C|$  (number of compassion requirements of  $\mathcal{D}_C$ ),  $k_A = |\mathcal{C}_A|$ ,  $n = |\Sigma_C| \cdot |\Sigma_A| \cdot (3^{k_C} + k_A)$ , and  $f = 2k_C + k_A$ .

**Theorem 1.** [HKR97,KV98] *We can solve fair simulation for  $\mathcal{D}_C$  and  $\mathcal{D}_A$  in time  $O(n^{2f+1} \cdot f!)$ .*

As we are interested in fair simulation as a precondition for trace inclusion, we take a more economic approach. Given two FDS's, we first convert the two to JDS's using the construction in Section 2. We then solve the simulation game for the two JDS's.

Consider the FDS's  $\mathcal{D}_C$  and  $\mathcal{D}_A$ . Let  $\mathcal{D}_C^B : \langle V_C^B, \mathcal{O}_C^B, \Theta_C^B, \rho_C^B, \mathcal{J}_C^B, \emptyset \rangle$  and  $\mathcal{D}_A^B : \langle V_A^B, \mathcal{O}_A^B, \Theta_A^B, \rho_A^B, \mathcal{J}_A^B, \emptyset \rangle$  be the JDS's equivalent to  $\mathcal{D}_C$  and  $\mathcal{D}_A$ . Consider the game  $G : \langle \mathcal{D}_C^B, \mathcal{D}_A^B \rangle$ . The winning condition for this game is:  $\bigwedge_{J_C \in \mathcal{J}_C^B} J_C \rightarrow \bigwedge_{J_A \in \mathcal{J}_A^B} J_A$ . We call such games *generalized Streett[1] games*. We claim that the formula in Equation (1) evaluates the set  $W_A$  of states winning for player  $A$ . Intuitively, the greatest fixpoint  $\nu X$  evaluates the set of states from which player  $A$  can control the run to remain in  $\neg J_k^C$  states. The least fixpoint  $\mu Y$  then evaluates the states from which player  $A$  in a finite number of steps controls the run to avoid one of the justice conditions  $J_k^C$ . This represents the set  $H$  of all states from which player  $A$  wins as a result of the run of  $\mathcal{D}_C^B$  violating justice. Finally, the outermost greatest fixpoint  $\nu Z_j$  adds to  $H$  the states from which player  $A$  can force the run to satisfy the fairness requirement of  $\mathcal{D}_A^B$ .

$$\varphi = \nu \begin{bmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_n \end{bmatrix} \left[ \begin{array}{l} \mu Y \left( \bigvee_{k=1}^m \nu X (J_1^A \wedge \otimes Z_2 \vee \otimes Y \vee \neg J_k^C \wedge \otimes X) \right) \\ \mu Y \left( \bigvee_{k=1}^m \nu X (J_2^A \wedge \otimes Z_3 \vee \otimes Y \vee \neg J_k^C \wedge \otimes X) \right) \\ \vdots \\ \mu Y \left( \bigvee_{k=1}^m \nu X (J_n^A \wedge \otimes Z_1 \vee \otimes Y \vee \neg J_k^C \wedge \otimes X) \right) \end{array} \right] \quad (1)$$

*Claim.*  $W_A = [[\varphi]]$

The proof of the claim will appear in the full version.

Using the algorithm in [EL86] the set  $[[\varphi]]$  can be evaluated symbolically.



**Theorem 2.** *The SGS  $G$  can be solved in time  $O((|\Sigma_C^B| \cdot |\Sigma_A^B| \cdot |\mathcal{J}_C^B| \cdot |\mathcal{J}_A^B|)^3)$ .*

To summarize, in order to use fair simulation as a precondition for trace inclusion we propose to convert the FDS's into JDS's and use the formula in Equation (1) to evaluate symbolically the winning set for player  $A$ .

**Corollary 1.** *Given  $\mathcal{D}_C$  and  $\mathcal{D}_A$ , we can determine whether  $\mathcal{D}_C^B \preceq_f \mathcal{D}_A^B$  in time proportional to  $O((|\Sigma_C| \cdot 2^{k_C} \cdot |\Sigma_A| \cdot 2^{k_A} \cdot (k_C + |\mathcal{J}_C| + k_A + |\mathcal{J}_A|))^3)$ .*

## 5 Closing the Gap

As discussed in the introduction, fair simulation implies trace inclusion but not the other way around. In [AL91], fair simulation is considered in the context of infinite-state systems. It is easy to see that the definition of *fair simulation* given in [AL91], is the infinite-state counterpart of fair simulation as defined in [HKR97], but restricted to memory-less strategies. As shown in [AL91], if we are allowed to add to the concrete system auxiliary *history* and *prophecy* variables, then the fair simulation method becomes complete for verifying trace inclusion.

Following [AL91], we allow the concrete system  $\mathcal{D}_C$  to be augmented with a set  $V_H$  of *history variables* and a set  $V_P$  of *prophecy variables*. We assume that the three sets,  $V_C$ ,  $V_H$ , and  $V_P$ , are pairwise disjoint. The result is an augmented concrete system  $\mathcal{D}_C^* : \langle V_C^*, \Theta_C^*, \rho_C^*, \mathcal{J}_C, \mathcal{C}_C \rangle$ , where

$$\begin{aligned} V_C^* &= V_C \cup V_H \cup V_P & \Theta_C^* &= \Theta_C \wedge \bigwedge_{x \in V_H} (x = f_x(V_C, V_P)) \\ \rho_C^* &= \rho_C \wedge \bigwedge_{x \in V_H} x' = g_x(V_C^*, V_C', V_P') \wedge \bigwedge_{y \in V_P} y = \varphi_y(V_C) \end{aligned}$$

In these definitions,  $f_x$  and  $g_x$  are state functions, while each  $\varphi_y(V_C)$  is a future temporal formula referring only to the variables in  $V_C$ . Thus, unlike [AL91], we use *transition relations* to define the values of history variables, and *future LTL formulas* to define the values of prophecy variables. The clause  $y = \varphi_y(V_C)$  added to the transition relation implies that at any position  $j \geq 0$ , the value of the boolean variable  $y$  is 1 iff the formula  $\varphi_y(V_C)$  holds at this position.

The augmentation scheme proposed above is *non-constraining*. Namely, for every computation  $\sigma$  of the original concrete system  $\mathcal{D}_C$  there exists a computation  $\sigma^*$  of  $\mathcal{D}_C^*$  such that  $\sigma$  and  $\sigma^*$  agree on the values of the variables in  $V_C$ .

Handling of the prophecy variables definitions is performed by constructing an appropriate *temporal tester* [KP00b] for each of the future temporal formulas appearing in the prophecy schemes, and composing it with the concrete system.

A similar augmentation of the concrete system has been used in [KPSZ02] in a deductive proof of abstraction, based on [AL91] *abstraction mapping*.

Although fair simulation is verified algorithmically, user intervention is still needed for choosing the appropriate temporal properties to be observed in order to ensure completeness with respect to trace inclusion.

We currently conjecture that we do not really need history variables, and we hope to prove this conjecture in a fuller version of this paper.

## 6 Examples

**Late and Early.** Consider, for example, the programs **EARLY** and **LATE** in Fig. 2 (graphic representation in Fig. 1). The observable variables are  $y$  and  $z$ . Wlog, assume that the initial values of all variables are 0. This is a well known example showing the difference between trace inclusion and simulation. Indeed, the two systems have the same set of traces. Either  $y$  assumes 1 or  $y$  assumes 2. On the other hand, **EARLY** does not simulate **LATE**. This is because we do not know whether the state  $\langle \ell_1, x:0, z:1 \rangle$  of system **LATE** should be mapped to state  $\langle \ell_1, x:1, z:1 \rangle$  or state  $\langle \ell_1, x:2, z:1 \rangle$  of system **EARLY**. Our algorithm shows that indeed **EARLY** does not simulate **LATE**.

EARLY ::	$\begin{bmatrix} \ell_0 : x, z := \{1, 2\}, 1 \\ \ell_1 : z := 2 \\ \ell_2 : y, z := x, 3 \end{bmatrix}$	LATE ::	$\begin{bmatrix} \ell_0 : z := 1 \\ \ell_1 : x, z := \{1, 2\}, 2 \\ \ell_2 : y, z := x, 3 \end{bmatrix}$
----------	--	---------	--

**Fig. 2.** Programs **EARLY** and **LATE**.

Since **EARLY** and **LATE** have the same set of traces, we should be able to augment **LATE** with prophecy variables that tell **EARLY** how to simulate it. In this case, we add a *tester*  $T_\varphi$  for the property  $\varphi : \Diamond(y = 1)$ . The tester introduces a new boolean variable  $x_\varphi$  which is true at a state  $s$  iff  $s \models \varphi$ . Whenever  $T_\varphi$  indicates that **LATE** will eventually choose  $x = 1$ , **EARLY** can safely choose  $x = 1$  in the first step. Whenever the tester for  $\Diamond(y = 1)$  indicates that **LATE** will never choose  $x = 1$ , **EARLY** can safely choose  $x = 2$  in the first step. Denote by  $\text{LATE}^+$  the combination of **LATE** with the tester  $\Diamond(y = 1)$ . Applying our algorithm to  $\text{LATE}^+$  and **EARLY**, indicates that  $\text{LATE}^+ \preceq_f \text{EARLY}$  implying  $\text{Obs}(\text{LATE}) \subseteq \text{Obs}(\text{EARLY})$ .

**Fair Discrete Modules and Open Computations.** For the main application of our technique, we need the notions of an *open system* and *open computations*.

We define a *fair discrete module* (FDM) to be a system  $M : \langle V, \mathcal{O}, W, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$  consisting of the same components as an FDS plus the additional component:

- $W \subseteq V$ : A set of *owned* variables. Only the system can modify these variables. All other variables can also be modified by steps of the environment.

An (*open*) *computation* of an FDM  $M$  is an infinite sequence  $\sigma : s_0, s_1, \dots$  of  $V$ -states which satisfies the requirements of initiality, justice, and compassion as any other FDS, and the requirement of consecution, reformulated as follows:

- **Consecution:** For each  $j = 0, 1, \dots$ ,
  - $s_{2j+1}[W] = s_{2j}[W]$ . That is,  $s_{2j+1}$  and  $s_{2j}$  agree on the interpretation of the owned variables  $W$ .
  - $s_{2j+2}$  is a  $\rho$ -successor of  $s_{2j+1}$ .

Thus, an (open) computation of an FDM consists of a strict interleaving of system with environment actions, where the system action has to satisfy  $\rho$ , while the environment step is only required to preserve the values of the owned variables.

Two FDM's  $\mathcal{D}_1$  and  $\mathcal{D}_2$  are compatible if  $W_1 \cap W_2 = \emptyset$  and  $V_1 \cap V_2 = O_1 \cap O_2$ . The asynchronous parallel composition of two compatible FDM's  $M = M_1 \parallel M_2$  is defined similarly to the case of composition of two FDS's where, in addition, the owned variables of the newly formed module is obtained as the union of  $W_{M_1}$  and  $W_{M_2}$ . Module  $M_2$  is said to be a *modular abstraction* of a comparable module  $M_1$ , denoted  $M_1 \sqsubseteq_M M_2$ , if  $Obs(M_1) \subseteq Obs(M_2)$ . A unique feature of the modular abstraction relation is that it is *compositional*, i.e.  $M_1 \sqsubseteq_M M_2$  implies  $M_1 \parallel M \sqsubseteq_M M_2 \parallel M$ . This compositionality allows us to replace a module  $M_1$  in any context of parallel composition by another module  $M_2$  which forms a modular abstraction of  $M_1$  and obtain an abstraction of the complete system, as needed in the network invariants method.

It is straightforward to reduce the problem of checking modular abstraction between modules to checking abstraction between FDS's using the methods presented in this paper. This reduction is based on a transformation which, for a given FDM  $M : \langle V, \mathcal{O}, W, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ , constructs an FDS  $\mathcal{D}_M : \langle \tilde{V}, \mathcal{O}, \tilde{\Theta}, \tilde{\rho}, \mathcal{J}, \mathcal{C} \rangle$ , such that the set of observations of  $M$  is equal to the set of observations of  $\mathcal{D}_M$ . The new components of  $\mathcal{D}_M$  are given by:

$$\begin{aligned} \tilde{V} &: V \cup \{t : \mathbf{boolean}\} & \tilde{\Theta} &: \Theta \wedge t \\ \tilde{\rho} &: \rho \wedge \neg t \wedge t' \quad \vee \quad pres(W) \wedge t \wedge \neg t' \end{aligned}$$

Thus, system  $\mathcal{D}_M$  uses a fresh boolean variable  $t$  to encode the turn taking between system and environment transitions.

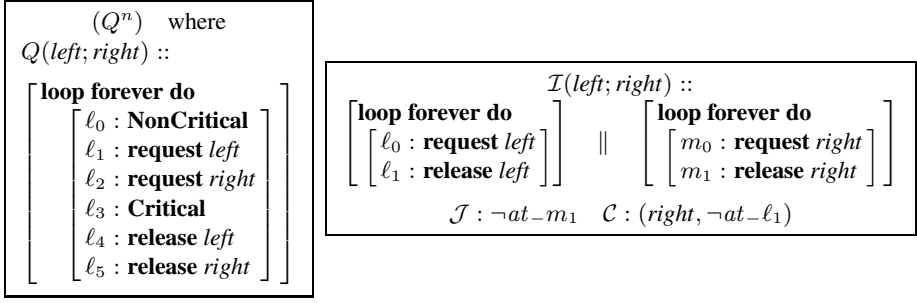
**The Dining Philosophers.** As a second example, we consider a solution to the dining philosophers problem. As originally described by Dijkstra,  $n$  philosophers are seated around a table, with a fork between each two neighbors. In order to eat a philosopher needs to acquire the forks on both its sides. A solution to the problem consists of protocols to the philosophers (and, possibly, forks) that guarantee that no two adjacent philosophers eat at the same time and that every hungry philosopher eventually eats.

A solution to the dining philosophers is presented in [KPSZ02], in terms of *binary processes*. A binary process  $Q(x; y)$  is an FDM with two observable variables  $x$  and  $y$ . Two binary processes  $Q$  and  $R$  can be composed to yield another binary process, using the *modular composition* operator  $\circ$  defined by

$$(Q \circ R)(x; z) = [\mathbf{restrict } y \text{ in } Q(x; y) \parallel R(y; z)]$$

where **restrict y** is an operator that removes variable  $y$  from the set of observable variables and places it in the set of owned variables.

In Fig. 3a we present a chain of  $n$  deterministic philosophers, each represented by a binary process  $Q(\text{left}; \text{right})$ . This solution is studied in [KPSZ02] as an example of parametric systems, for which we seek a *uniform* verification (i.e. a single verification valid for any  $n$ ). The uniform verification is presented using the network invariant method, which calls for the identification of a network invariant  $\mathcal{I}$  which can safely replace the chain  $Q^n$ . The adequacy of the network invariant is verified using an inductive argument which calls for the verification of abstractions. In [KPSZ02] we present a deductive proof to the dining philosophers, based on [AL91] abstraction mapping method, using two different network invariants.



**Fig. 3.** (a) Program DINE. (b) the two halves abstraction.

Here, we consider the same invariants, and verify all the necessary abstractions using our algorithm for fair simulation. In both cases, no auxiliary variables are needed.

**The “Two-Halves” Abstraction.** The first network invariant  $\mathcal{I}(left; right)$  is presented in Fig. 3b and can be viewed as the parallel composition of two “one-sided” philosophers. The compassion requirement reflects the fact that  $\mathcal{I}$  can deadlock at location  $\ell_1$  only if, from some point on, the fork on the right (*right*) is continuously unavailable.

To establish that  $\mathcal{I}$  is a network invariant, we verify the abstractions  $(Q \circ Q) \sqsubseteq_M \mathcal{I}$  and  $(Q \circ \mathcal{I}) \sqsubseteq_M \mathcal{I}$  using the fair simulation algorithm.

**The “Four-by-Three” Abstraction.** An alternative network invariant is obtained by taking  $\mathcal{I} = Q^3$ , i.e. a chain of 3 philosophers. To prove that this is an invariant, it is sufficient to establish the abstraction  $Q^4 \sqsubseteq_M Q^3$ , that is, to prove that 3 philosophers can faithfully emulate 4 philosophers.

**Experimental Results.** We include in our implementation the following steps:

- Removal of all unfeasible states from both systems. Thus, the first step evaluates the set of feasible states for each system [KPR98].
- Recall that fair simulation implies simulation [HKR97]. Let  $S \subseteq \Sigma_{cor}$  denote the maximal simulation relation. To optimize the algorithm we further restrict player  $A$ ’s moves to  $S$  instead of  $\Sigma_{cor}$ .

The following summarizes the running time for some of our experiments.

$(Q \circ Q) \sqsubseteq_M \mathcal{I}$	44 secs.	$(Q \circ \mathcal{I}) \sqsubseteq_M \mathcal{I}$	6 secs.	$Q^4 \sqsubseteq_M Q^3$	178 secs.
---	----------	---	---------	-------------------------	-----------

## 7 Acknowledgments

We thank Orna Kupferman for suggesting using fair simulation for algorithmically verifying abstraction of reactive systems.

## References

- [AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *TCS*, 82, 1991.
- [BR96] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *SCP*, 24:189–220, 1996.
- [Cho74] Y. Choueka. Theories of automata on  $\omega$ -tapes: A simplified approach. *JCSS*, 1974.
- [dAHM01] L. de Alfaro, T.A. Henzinger, and R. Majumdar. From verification to control: dynamic programs for omega-regular objectives. In *16th LICS*, 2001.
- [EL86] E. A. Emerson and C. L. Lei. Efficient model-checking in fragments of the propositional modal  $\mu$ -calculus. In *1st LICS*, 267–278, 1986.
- [Eme97] E.A. Emerson. Model checking and the  $\mu$ -calculus. In *Descriptive Complexity and Finite Models*, pages 185–214. AMS, 1997.
- [GH82] Y. Gurevich and L.A. Harrington. Automata, trees and games. In *14th STOC*, 1982.
- [GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.
- [HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *36th FOCS*, 453–462, 1995.
- [HKR97] T.A. Henzinger, O. Kupferman, and S. Rajamani. Fair simulation. In *8th CONCUR*, LNCS 1243, 273–287, 1997.
- [HR00] T. Henzinger and S. Rajamani. Fair bisimulation. In *6th TACAS* LNCS 1785, 2000.
- [Jur00] M. Jurdzinski. Small progress measures for solving parity games. In *17th STACS*, LNCS 1770, 290–301, 2000.
- [Koz83] D. Kozen. Results on the propositional  $\mu$ -calculus. *TCS*, 27:333–354, 1983.
- [KP00a] Y. Kesten and A. Pnueli. Control and data abstractions: The cornerstones of practical formal verification. *STTT*, 2(1):328–342, 2000.
- [KP00b] Y. Kesten and A. Pnueli. Verification by finitary abstraction. *IC*, 163:203–243, 2000.
- [KPR98] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *25th ICALP*, LNCS 1443, 1–16. 1998.
- [KPSZ02] Y. Kesten, A. Pnueli, E. Shahar, and L. Zuck. Network invariants in action. In *13th CONCUR*, LNCS 2421, 101–105, 2002.
- [KPV00] O. Kupferman, N. Piterman, and M.Y. Vardi. Fair equivalence relations. In *20th FSTTCS*, LNCS 1974, 151–163. 2000.
- [KV98] O. Kupferman and M.Y. Vardi. Weak alternating automata and tree automata emptiness. In *30th STOC*, 224–233, 1998.
- [LT87] K. Lodaya and P.S. Thiagarajan. A modal logic for a subclass of events structures. In *14th ICALP*, LNCS 267, 290–303, 1987.
- [MAB<sup>+</sup>94] Z. Manna, A. Anuchitanukul, N. Bjørner, A. Browne, E. Chang, M. Colón, L. De Alfaro, H. Devarajan, H. Sipma, and T.E. Uribe. STeP. Tech. Report, Stanford 1994.
- [Mil71] R. Milner. An algebraic definition of simulation between programs. *IJCAI*, 1971.
- [MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
- [Saf92] S. Safra. Exponential determinization for  $\omega$ -automata with strong-fairness acceptance condition. In *24th STOC*, 1992.
- [Var91] M. Y. Vardi. Verification of concurrent programs – the automata-theoretic framework. *APAL*, 51:79–98, 1991.
- [WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *AVMFS*, LNCS 407, 68–80. 1989.

# An Improved On-the-Fly Tableau Construction for a Real-Time Temporal Logic

Marc Geilen

Dept. of Elec. Eng., Eindhoven University of Technology, The Netherlands  
m.c.w.geilen@tue.nl

**Abstract.** Temporal logic is popular for specifying correctness properties of reactive systems. Real-time temporal logics add the ability to express quantitative timing aspects. Tableau constructions are algorithms that translate a temporal logic formula into a finite-state automaton that accepts precisely all the models of the formula. On-the-fly versions of tableau-constructions enable their practical application for model-checking. In a previous paper we presented an on-the-fly tableaux construction for a fragment of Metric Interval Temporal Logic in dense time. The interpretation of the logic was constrained to a special kind of timed state sequences, with intervals that are always left-closed and right-open. In this paper we present a non-trivial extension to this tableau construction for the same logic fragment, that lifts this constraint.

## 1 Introduction

Temporal logic [13] is a popular formalism for expressing properties of reactive systems. Finite state models of such programs can be analysed automatically with model checking procedures. Temporal logics exist in different flavours. In particular, *real-time* temporal logics have been developed for expressing timing aspects of systems. A *tableau construction* is an algorithm that translates a temporal logic formula into a finite-state automaton that accepts precisely all the models of the formula. The *automata-theoretic* approach to model checking [12,14] employs tableau algorithms to turn a temporal formula into an observer of a system's behaviours. For practical applications, tableau constructions are being improved and optimised (e.g., [5,6,9]). An important improvement has been the development of *on-the-fly* versions of tableau constructions. Such on-the-fly tableau constructions are based on a normal form for temporal formulas in which the constraints on the current state are separated from the constraints on the future states. In the real-time domain, tableau constructions have been developed for various logics and their complexities have been studied [2,10]. For dense time linear temporal logic the algorithms of [2,4] and [8] are (to the best of the author's knowledge) the only existing tableau constructions to date. [2,4] being aimed at establishing a theoretical connection between temporal logic and timed automata and [8] a first attempt at a practical algorithm suitable for model checking. [1] describes the construction of so-called testing automata from

formulas in a safety modal logic for the UPPAAL tool. The logic is a restricted variant of a real-time extension of Hennessy-Milner logic.

Towards the creation of an efficient tableau construction for dense real-time, we presented in [8] a first version of an on-the-fly tableau construction for a linear temporal logic with dense time. A drawback of this work is that formulas are interpreted over a restricted kind of state sequences. The logic considered both in that paper and this paper, is based on a fragment of *Metric Interval Temporal Logic* (MITL, see [4]) that is decidable in PSPACE. In this paper we remove the restriction of [8] and show that, at the expense of some extra bookkeeping and hence some more locations in the tableaux, we can apply the similar ideas to obtain an on-the-fly tableau construction for a dense-time linear temporal logic in the traditional interpretation, enabling the connection to common theory and tools. We also demonstrate a small flaw in the classical result of [2,4], a tableau construction for MITL, that can be solved with techniques introduced in this paper. The new tableau algorithm is also based on a normal form procedure for temporal logic formulas, that separates constraints on the current interval from constraints on the remainder of the state sequence. In order to deal with arbitrary interval types, two normal forms are considered related to two kinds of time intervals, singular and open intervals. In order to define these normal forms, the logic is extended with timers that can be explicitly set and tested, and with a *Next* operator referring to the beginning of the next time interval.

Section 2 introduces the logic and timed automata that we use. Section 3 discusses the main ideas behind the tableau construction. Section 4 presents the normal form for formulas on which the tableau algorithm of Section 5 is based. Section 6 concludes.

## 2 Preliminaries

An  $(\omega\text{-})$ word  $\bar{w} = \sigma_0\sigma_1\sigma_2\dots$  over an alphabet  $\Sigma$  is an infinite sequence of symbols from  $\Sigma$ ;  $\bar{w}(k)$  denotes  $\sigma_k$  and  $\bar{w}^k$  refers to the *tail*  $\sigma_k\sigma_{k+1}\sigma_{k+2}\dots$ . We use the latter notations for other kinds of sequences as well. An *interval*  $I$  is a convex subset of  $\mathbb{R}^{\geq 0}$ ; it has one of four shapes:  $[a, b]$ ,  $[a, b)$ ,  $(a, b]$  or  $(a, b)$ .  $l(I)$  ( $r(I)$ ) denotes the lower (upper) bound  $a$  ( $b$ ) and  $|I|$  the length of  $I$ . We use  $I - t$  to denote the interval  $\{t' - t \in \mathbb{R}^{\geq 0} \mid t' \in I\}$ . An *interval sequence*  $\bar{I} = I_0I_1I_2\dots$  is an infinite sequence of intervals that are *adjacent*, meaning  $r(I_i) = l(I_{i+1})$  and that are of matching type (right-closed and left-open or right-open and left-closed) for every  $i$ , for which  $l(I_0) = 0$ , and which is *diverging*, i.e. any  $t \in \mathbb{R}^{\geq 0}$  belongs to precisely one interval  $I_i$ . A *timed word*  $\bar{u}$  over  $\Sigma$  is a pair  $(\bar{w}, \bar{I})$  consisting of an  $\omega$ -word  $\bar{w}$  over  $\Sigma$  and an interval sequence  $\bar{I}$ . For  $t \in \mathbb{R}^{\geq 0}$ ,  $\bar{u}(t)$  denotes the symbol present at time  $t$ , this is  $\bar{w}(k)$  if  $t \in \bar{I}(k)$ . For such a  $t$  and  $k$ ,  $\bar{u}^t$  is the tail of the timed word, consisting of the word  $\bar{w}^k$  and of the interval sequence  $\bar{I}(k) - t$ ,  $\bar{I}(k+1) - t, \dots$ . The timed word determines a mapping from  $\mathbb{R}^{\geq 0}$  to  $\Sigma$ . Timed words  $\bar{u}_1$  and  $\bar{u}_2$  are *equivalent* (denoted  $\bar{u}_1 \equiv \bar{u}_2$ ) if they define the same mapping, i.e. for all  $t \geq 0$ ,  $\bar{u}_1(t) = \bar{u}_2(t)$ . A *timed state sequence* (TSS) over a set *Prop* of propositions is a timed word over the alphabet  $2^{\text{Prop}}$ .

## 2.1 Real-Time Temporal Logic

We consider a restricted version of the real-time temporal logic MITL of [4],  $\text{MITL}_{\leq}$ , with formulas of the following form ( $d \in \mathbb{N}$ ).

$$\varphi ::= \text{true} \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_{\leq d} \varphi_2$$

Formulas of this form are called *basic*, in order to distinguish them from formulas using an extended syntax that is to be defined in Section 4.1. A formula is interpreted over a timed state sequence  $\bar{\rho}$  as follows.

$$\begin{aligned} \bar{\rho} \models \text{true} & \quad \text{for every timed state sequence } \bar{\rho}; \\ \bar{\rho} \models p & \quad \text{iff } p \in \bar{\rho}(0); \\ \bar{\rho} \models \neg\varphi & \quad \text{iff not } \bar{\rho} \models \varphi; \\ \bar{\rho} \models \varphi_1 \vee \varphi_2 & \quad \text{iff } \bar{\rho} \models \varphi_1 \text{ or } \bar{\rho} \models \varphi_2; \\ \bar{\rho} \models \varphi_1 \mathbf{U}_{\leq d} \varphi_2 & \quad \text{iff there is some } 0 \leq t \leq d, \text{ such that } \bar{\rho}^t \models \varphi_2 \text{ and} \\ & \quad \text{for all } 0 \leq t' < t, \bar{\rho}^{t'} \models \varphi_1. \end{aligned}$$

Note that no basic  $\text{MITL}_{\leq}$  formula can distinguish equivalent timed state sequences. We also use the duals:  $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$ , and Release  $\varphi_1 \mathbf{V}_{\leq d} \varphi_2 = \neg(\neg\varphi_1 \mathbf{U}_{\leq d} \neg\varphi_2)$ . The restriction of the bound  $d$  to finite naturals simplifies the presentation as it yields a logic in which only safety properties can be expressed, thus avoiding the need to handle acceptance conditions. It is straightforward to extend our results using standard techniques (see e.g., [9]) to a logic which also includes an unbounded Until operator. Also other types of bounds ( $< d$ ,  $\geq d$ ,  $> d$ ) should be easy to add.

**$\varphi$ -Fine Interval Sequences** In the on-the-fly tableau constructions of [5,9] for untimed logics, a formula is rewritten so that the constraints on the current state are separated from those on the remainder of the state sequence. This is possible because of the discrete nature of the sequence. In the dense time case, there is no such thing as a next state. The discretisation suggested by the interval sequence  $\bar{I}$  of a timed state sequence  $\bar{\rho}$  is, in general, not fine enough for our purposes: when interpreted over tails of  $\bar{\rho}$ , the truth value of a formula may vary along a single interval of  $\bar{I}$ .

**Definition 1.** ([2,8]) Let  $\varphi \in \text{MITL}_{\leq}$ . An interval sequence  $\bar{I}$  is called  $\varphi$ -fine for timed state sequence  $\bar{\rho}$  if for every syntactic subformula  $\psi$  of  $\varphi$ , every  $k \geq 0$ , and every  $t_1, t_2 \in \bar{I}(k)$ , we have  $\bar{\rho}^{t_1} \models \psi$  iff  $\bar{\rho}^{t_2} \models \psi$ . In case that  $\bar{I}$  is  $\varphi$ -fine for a timed state sequence  $(\bar{\sigma}, \bar{I})$ , also  $(\bar{\sigma}, \bar{I})$  will be called  $\varphi$ -fine.

In [2], Lemma 4.11 it is shown that the intervals of any timed state sequence can always be refined to be fine for any MITL formula. Being a subset of MITL, the same holds for  $\text{MITL}_{\leq}$ . Since every refinement (by splitting intervals) of a  $\varphi$ -fine interval sequence is again a  $\varphi$ -fine interval sequence, it follows that there must also exist a  $\varphi$ -fine interval sequence consisting of only singular and open intervals [2]. In the remainder of the paper we will assume (without loss of generality) that every interval sequence  $\bar{I}$  consists of alternating singular and open intervals, i.e.  $\bar{I}(0)$  is singular,  $\bar{I}(1)$  is open,  $\bar{I}(2)$  is singular, and so on.



## 2.2 Timed Automata

The target of our tableau construction are timed automata in the style of Alur and Dill [3]. We use a variation, adapted to our needs, that can easily be translated to Alur-Dill automata. We would like to control the type of interval transitions (open to closed or closed to open). Therefore, we use two kinds of locations, locations ( $L_s$ ) in which the automaton may only reside for a single instant and locations ( $L_o$ ) in which the automaton may only reside for intervals that are both left-open and right-open. The automaton must necessarily alternate both kinds of locations. This behaviour can be easily enforced in a standard timed automaton (see e.g., [2]). A location can only be visited in singular intervals if a timer is set to 0 upon entering the location and is required to be equal to 0 while staying in that location. Locations visited in between singular intervals can then only correspond to open intervals. Moreover, the automata use *timers* that decrease as time advances, possibly taking negative values in  $\mathbb{R}$  (this is to facilitate the correspondence with temporal logic formulas). They may be set to nonnegative integer values and may be compared to zero only. Given a set  $T$  of timers, a *timer valuation*  $\nu \in TVal(T)$  is a mapping  $T \rightarrow \mathbb{R}$ . For  $t \in \mathbb{R}^{\geq 0}$ ,  $\nu - t$  denotes the timer valuation that assigns  $\nu(x) - t$  to any timer  $x$  in the domain of  $\nu$ . A *timer setting*  $TS \in TSet(T)$  is a partial mapping  $T \rightarrow \mathbb{N}$ . We use  $TS[x := d]$  (where  $x \in T$  and  $d \in \mathbb{N}$ ) to denote the timer setting that maps  $x$  to  $d$  and other timers to the same value as  $TS$ .  $[x := d]$  is short for  $\emptyset[x := d]$ . For a timer valuation  $\nu$  and a timer setting  $TS$ ,  $TS(\nu)$  is the timer valuation that maps any timer  $x$  in the domain of  $\nu$  to  $TS(x)$  if defined, and to  $\nu(x)$  otherwise. The set  $TCond(T)$  of *timer conditions over  $T$*  is  $\{x > 0, x \geq 0, x < 0, x \leq 0 \mid x \in T\}$ .

**Definition 2.** Let  $\Sigma$  be an alphabet. A timed automaton  $A = \langle L_s, L_o, T, L_0, Q, TC, E \rangle$  over  $\Sigma$  consists of

- a finite set  $L = L_s \cup L_o$  of locations  $\ell$  that are either singular ( $\ell \in L_s$ ) or open ( $\ell \in L_o$ );
- a set  $T$  of timers;
- a finite set  $L_0$  of initial extended locations  $(\ell_0, \nu_0) \in L_s \times TVal(T)$ , where  $\nu_0$  assigns integer values to the timers;
- a mapping  $Q : L \rightarrow 2^\Sigma$  labelling every location with a set of symbols from the alphabet;
- a mapping  $TC : L \rightarrow 2^{TCond(T)}$  labelling every location with a set of timer conditions over  $T$ ;
- a set  $E \subseteq L \times TSet(T) \times L$  of edges labelled by timer settings, such that for every edge  $\langle \ell, TS, \ell' \rangle \in E$ , if  $\ell \in L_s$  then  $\ell' \in L_o$  and if  $\ell \in L_o$  then  $\ell' \in L_s$ .

An *extended location*  $\lambda$  is a pair  $(\ell, \nu)$  consisting of a location  $\ell$  and a timer valuation  $\nu$ . For an automaton with the set  $T$  of timers, we use  $\bar{0}$  to denote the timer valuation that maps every timer in  $T$  to 0. A *timed run* describes the path taken by the automaton when accepting a timed word. It gives the location of the automaton and the values of its timers at any moment, by recording the sequence of locations, the intervals during which the automaton resides in those locations, and the timer values at the time of transition to the interval.

**Definition 3.** A timed run  $\bar{r}$  of an automaton  $\langle L_s, L_o, T, L_0, Q, TC, E \rangle$  is a triple  $(\bar{\ell}, \bar{I}, \bar{\nu})$  consisting of a sequence of locations, an interval sequence, alternating singular and open intervals, and a sequence of timer valuations, such that:

- [Consecution] for all  $k \geq 0$ , there is an edge  $(\bar{\ell}(k), TS_k, \bar{\ell}(k+1)) \in E$  such that  $\bar{\nu}(k+1) = TS_k(\bar{\nu}(k) - |\bar{I}(k)|)$ ;
- [Timing] for all  $k \geq 0$  and  $t \in \bar{I}(k)$ , the timer valuation at time  $t$ ,  $\bar{\nu}(k) - (t - l(\bar{I}(k)))$ , satisfies all timer conditions in  $TC(\bar{\ell}(k))$ .

We write  $\bar{r}(t)$  to denote the location of  $\bar{r}$  at time  $t$ , i.e.  $\bar{\ell}(k)$  if  $t \in \bar{I}(k)$ . Given a timed word  $\bar{u}$ ,  $\bar{r}$  is a run for  $\bar{u}$  if

- [Symbol match] for all  $t \geq 0$ ,  $\bar{u}(t) \in Q(\bar{r}(t))$ .

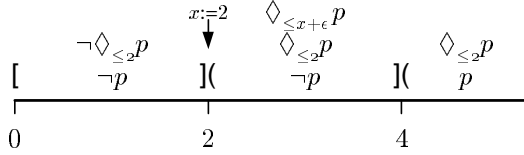
A accepts  $\bar{u}$  if it has a run for  $\bar{u}$  from an initial extended location. The timed language  $\mathcal{L}(A)$  of  $A$  is the set of all timed words that it accepts.

It follows from Def. 3 that the language of a timed automaton is closed under equivalence. This will allow us to restrict our attention to  $\varphi$ -fine sequences, alternating singular and open intervals, when arguing the completeness of the tableau construction for a basic MITL $_{\leq}$  formula  $\varphi$ .

### 3 Towards a Tableau Construction

In the tableau construction, we are going to use timers to remember when certain relevant transitions have been made and to enforce time constraints between such transitions. Unfortunately, recording the time of transition alone is not enough, it is also necessary to remember in what fashion the transition was performed, from a right-open to a left-closed interval or from a right-closed to a left-open one. This can be demonstrated using Fig. 1, we see the beginning of a timed state sequence with interval sequence  $[0, 2] (2, 4] (4, \dots$  and state sequence  $\emptyset \emptyset \{p\} \dots$  and the formula  $\Diamond_{\leq 2} p$  (short for  $\text{true} \cup_{\leq 2} p$ ). It is easy to check that the first intervals of the timed state sequence are  $\Diamond_{\leq 2} p$ -fine and that  $\Diamond_{\leq 2} p$  does not hold in the first interval and does hold in the second and third. If we imagine the tableau automaton trying to accept this TSS and checking that  $\Diamond_{\leq 2} p$  holds in the second interval (which is left-open), it would need to set a timer, say  $x$ , at time 2, the time of the transition. Now it must check that a state is reached where  $p$  holds within 2 units of time. To be precise, as it is the case in Fig. 1,  $p$  must hold at the latest in some left-open interval starting at time instant 4. This would have been different, however, if the transition at time 2 were from a right-open to a left-closed interval, in that case,  $p$  must hold at the latest at time instant 4 itself. This example shows that one must record not only the time of the transition, but also its type. To deal with this issue,  $[2, 4]$  distinguishes singular and open intervals. However, we believe that the tableau construction of  $[2, 4]$  does not deal with the former situation correctly, in this case it requires only that  $p$  holds at the latest at the instant 4 itself and does not allow for  $p$  to become true immediately after that<sup>1</sup>; this can be resolved using the technique outlined in this paper.

<sup>1</sup> Assume (using notation of  $[2, 4]$ ) we have the MITL formula  $\psi = \Diamond_{(0, 1]} p$ . Consider an incoming edge to an open-interval location containing  $\psi$  from a location not



**Fig. 1.** Checking the formula  $\Diamond_{\leq 2p}$

Fig. 1 also shows how a tableau automaton can deal with checking the formula  $\Diamond_{\leq 2p}$ . At the beginning of the interval, a timer  $x$  is set to 2. The type of transition is recorded in the form of the constraint that is remembered during the subsequent interval. This constraint,  $\Diamond_{\leq x+\epsilon} p$ , represents the fact that during the interval (while  $x$  decreases), for any  $\epsilon > 0$ , there must be some point in time, at most  $x + \epsilon$  units of time away, where  $p$  holds. We introduce a number of constructs that we will later learn how to interpret as formulas themselves and are needed to define the tableau automata.

**Definition 4.** With an *Until* formula  $\psi = \varphi_1 \mathbf{U}_{\leq d} \varphi_2$  we associate the set:

$$\Xi^\psi = \{\varphi_1 \mathbf{U}_{\leq x_\psi} \varphi_2, \varphi_1 \mathbf{U}_{\leq x_\psi + \epsilon} \varphi_2, x_\psi > 0, x_\psi \geq 0\}$$

and with a *Release* formula  $\psi = \varphi_1 \mathbf{V}_{\leq d} \varphi_2$  we associate the set:

$$\Xi^\psi = \{\varphi_1 \mathbf{V}_{< d} \varphi_2, \varphi_1 \mathbf{V}_{< x_\psi} \varphi_2, \varphi_1 \mathbf{V}_{\leq x_\psi} \varphi_2, x_\psi < 0, x_\psi \leq 0\}.$$

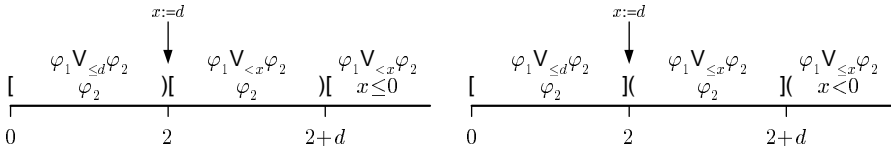
The closure  $cl(\varphi)$  of an MITL $_{\leq}$  formula  $\varphi$  in positive form is the smallest set  $\Phi$  such that

- every syntactic subformula of  $\varphi$ , including  $\varphi$  itself, is in  $cl(\varphi)$ ;
- for every *Until* or *Release* formula  $\psi \in cl(\varphi)$ ,  $\Xi^\psi \subseteq cl(\varphi)$ .

In untimed tableau constructions, the constraints on the current state are separated from the constraints on the remainder of the state sequence. In the tableau for real-time temporal logic with only one kind of intervals of [8], we mimicked this by separating constraints on the first interval from constraints on the timed state sequence starting from the second interval. In this case, we try the same. However, the constraints that need to be imposed on the next interval now depend on the type of transition as well. Fig. 2 shows another example. If we have to check that  $\varphi_1 \mathbf{V}_{\leq d} \varphi_2$  holds during some initial interval of length 2, and we know that  $\varphi_2$  holds during this interval but we do not know that  $\varphi_1$  holds, then at the end of the interval we need to set a timer to  $d$  and check that  $\varphi_2$  remains

---

containing  $x_\psi$ . Then the timer is allowed to be reset upon entering the location. From that point onwards, subsequent locations contain  $x_\psi$  and the constraint  $x_\psi \in (0, 1]$  until a location is reached containing  $p$  and  $x_\psi \in (0, 1]$  and the timer  $x_\psi$  is not reset in the mean time. Thus by the time when  $x_\psi = 1$ , a location containing  $p$  must have been reached, otherwise the run is stuck. But the formula holds in the initial open interval if  $p$  becomes true immediately after this moment.



**Fig. 2.** Separating constraints for the next interval depending on transition type

to hold until either the timer expires or  $\varphi_1$  holds. How long exactly,  $\varphi_2$  needs to hold depends again on the type of transition. If it is from a right open to a left closed interval (Fig. 2, left), then the formula  $\varphi_1 V_{\leq d} \varphi_2$  need not hold at the time of the transition itself and therefore,  $\varphi_2$  is only required to hold up to but not including the point where the timer expires, hence the constraint  $\varphi_1 V_{< x} \varphi_2$ . If the transition is from a right-closed to a left open interval (see Fig. 2, right), then the formula  $\varphi_1 V_{\leq d} \varphi_2$  must also hold at the time of the transition and  $\varphi_2$  must still hold at the point where the timer expires, hence the constraint  $\varphi_1 V_{\leq x} \varphi_2$ .

## 4 A Disjunctive Temporal Normal Form

On-the-fly tableau constructions are generally based on rewriting formulas in some disjunctive temporal normal form that separates constraints on the current moment from constraints on the future and that orders disjunction and conjunction so that disjunctions can be mapped to non-deterministic choice of the automaton. In [8], we used intervals for the discretisation of the time-domain that allows to effectively separate ‘now’ from ‘the future’. For this we introduced a  $\bigcirc$  operator that supported this for the type of intervals that we used. Now we want to repeat this approach for the interval sequences of alternatingly singular and open intervals. We essentially present two normal form rewrite procedures, one for each interval type, that preserve equivalence of the formula throughout that interval. To this end, we start with two satisfaction relations. The first, denoted  $\models^s$ , for singular intervals, is in fact the ordinary satisfaction relation  $\models$ , but is furthermore defined for the extended formulas introduced in the next section. The second,  $\models^o$ , asserts that a formula holds during the first open interval.

### 4.1 Extending the Logic

In order to define the normal form towards the construction of timed automata, the logic is extended with timers that can be bound by timer setting operators or which can be free. A formula is interpreted in the context of a timer environment which provides a value for every free timer. These timers will be related to timers of the timed tableau automaton. Besides that, we introduce a Next operator ( $\bigcirc$ ), that will form the basis for deriving its transitions. Moreover we assume that formulas are in positive normal form (negations only occur in front of propositions) using the dual operators (false,  $\wedge$ ,  $\vee$ ). The timer set operator

can set timers to integer values, and timers can be compared to 0 only by checking the conditions  $x > 0$ ,  $x \geq 0$ ,  $x < 0$  or  $x \leq 0$ . The arguments of an Until or Release operator must be basic MITL<sub>≤</sub> formulas.

MITL<sub>≤</sub> is redefined. Besides the basic formulas defined in section 2 we add formulas of the following form, where  $\varphi$ ,  $\varphi_1$ , and  $\varphi_2$  are basic formulas,  $d \in \mathbb{N}$ ,  $TS$  is a timer setting,  $x$  is a timer and  $\sim \in \{<, \leq, >, \geq\}$ .

$$\begin{aligned} \psi ::= & \varphi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid TS.\psi \mid x \sim 0 \mid \varphi_1 \mathbf{V}_{<d}\varphi_2 \mid \varphi_1 \mathbf{U}_{\leq x}\varphi_2 \mid \varphi_1 \mathbf{U}_{\leq x+\epsilon}\varphi_2 \\ & \mid \varphi_1 \mathbf{V}_{<x}\varphi_2 \mid \varphi_1 \mathbf{V}_{\leq x}\varphi_2 \mid \bigcirc \psi. \end{aligned}$$

The semantics is extended as follows. We write  $\bar{\rho} \models_{\nu}^s \psi$  to denote that the timed state sequence  $\bar{\rho}$  satisfies  $\psi$  in the context of the timer valuation  $\nu$ .

$$\begin{aligned} \bar{\rho} \models_{\nu}^s \varphi & \quad \text{iff} \quad \bar{\rho} \models \varphi; \\ \bar{\rho} \models_{\nu}^s \psi_1 \vee \psi_2 & \quad \text{iff} \quad \bar{\rho} \models_{\nu}^s \psi_1 \text{ or } \bar{\rho} \models_{\nu}^s \psi_2; \\ \bar{\rho} \models_{\nu}^s \psi_1 \wedge \psi_2 & \quad \text{iff} \quad \bar{\rho} \models_{\nu}^s \psi_1 \text{ and } \bar{\rho} \models_{\nu}^s \psi_2; \\ \bar{\rho} \models_{\nu}^s \varphi_1 \mathbf{V}_{<d}\varphi_2 & \quad \text{iff} \quad \text{for all } 0 \leq t < d, \bar{\rho}^t \models_{\nu-t}^s \varphi_2 \text{ or there is some } 0 \leq t' < t, \text{ such that } \bar{\rho}^{t'} \models_{\nu-t'}^s \varphi_1; \\ \bar{\rho} \models_{\nu}^s TS.\psi & \quad \text{iff} \quad \bar{\rho} \models_{TS(\nu)}^s \psi; \\ \bar{\rho} \models_{\nu}^s \varphi_1 \mathbf{U}_{\leq x}\varphi_2 & \quad \text{iff} \quad \bar{\rho} \models_{\nu}^s \varphi_2 \text{ or there is some } 0 \leq t \leq \nu(x), \text{ such that } \bar{\rho}^t \models_{\nu-t}^s \varphi_2 \text{ and for all } 0 \leq t' < t, \bar{\rho}^{t'} \models_{\nu-t'}^s \varphi_1; \\ \bar{\rho} \models_{\nu}^s \varphi_1 \mathbf{U}_{\leq x+\epsilon}\varphi_2 & \quad \text{iff} \quad \bar{\rho} \models_{\nu}^s \varphi_2 \text{ or for every } \epsilon > 0, \text{ there is some } 0 \leq t \leq \nu(x) + \epsilon, \text{ such that } \bar{\rho}^t \models_{\nu-t}^s \varphi_2 \text{ and for all } 0 \leq t' < t, \bar{\rho}^{t'} \models_{\nu-t'}^s \varphi_1; \\ \bar{\rho} \models_{\nu}^s \varphi_1 \mathbf{V}_{<x}\varphi_2 & \quad \text{iff} \quad \text{for all } 0 \leq t < \nu(x), \bar{\rho}^t \models_{\nu-t}^s \varphi_2 \text{ or there is some } 0 \leq t' < t, \text{ such that } \bar{\rho}^{t'} \models_{\nu-t'}^s \varphi_1; \\ \bar{\rho} \models_{\nu}^s \varphi_1 \mathbf{V}_{\leq x}\varphi_2 & \quad \text{iff} \quad \text{for all } 0 \leq t \leq \nu(x), \bar{\rho}^t \models_{\nu-t}^s \varphi_2 \text{ or there is some } 0 \leq t' < t, \text{ such that } \bar{\rho}^{t'} \models_{\nu-t'}^s \varphi_1; \\ \bar{\rho} \models_{\nu}^s x \sim 0 & \quad \text{iff} \quad \nu(x) \sim 0; \\ \bar{\rho} \models_{\nu}^s \bigcirc \psi & \quad \text{iff} \quad \bar{\rho} \models_{\nu}^o \psi \end{aligned}$$

$\models^o$  in the last clause is defined for all formulas, except  $TS.\psi$  and  $\bigcirc \psi$ , as:

$$\bar{\rho} \models_{\nu}^o \psi \quad \text{iff} \quad \text{for every } t \in \bar{I}(1), \bar{\rho}^t \models_{\nu-t}^s \psi$$

Moreover,

$$\begin{aligned} \bar{\rho} \models_{\nu}^o TS.\psi & \quad \text{iff} \quad \bar{\rho} \models_{TS(\nu)}^o \psi; \\ \bar{\rho} \models_{\nu}^o \bigcirc \psi & \quad \text{iff} \quad \bar{\rho}^{l(\bar{I}(2))} \models_{\nu-l(\bar{I}(2))}^s \psi; \end{aligned}$$

Notice, in particular, the Until formula with bound  $\leq x + \epsilon$ , which arises from the need to check a bounded Until formula during an entire (left-open) interval as discussed in the example in Section 3. In the remainder we use the notation  $\stackrel{it}{\equiv}$ ,  $it \in \{s, o\}$  to denote the following.  $\psi_1 \stackrel{it}{\equiv} \psi_2$  iff for every  $\psi_1$ -fine and  $\psi_2$ -fine timed state sequence  $\bar{\rho}$ , and every  $\nu$ ,  $\bar{\rho} \models_{\nu}^{it} \psi_1 \iff \bar{\rho} \models_{\nu}^{it} \psi_2$ . If conditions on the timer valuation  $\nu$  are mentioned besides the expression  $\psi_1 \stackrel{it}{\equiv} \psi_2$ , it means that satisfaction is only identical for timer valuations  $\nu$  that satisfy those criteria. We also write  $\psi_1 \stackrel{s,o}{\equiv} \psi_2$  to express that both  $\psi_1 \stackrel{s}{\equiv} \psi_2$  and  $\psi_1 \stackrel{o}{\equiv} \psi_2$ .

The goal is to rewrite  $\text{MITL}_{\leq}$  formulas into disjunctive temporal normal form, i.e., into a formula of the form

$$\bigvee_i TS_i.(\pi_i \wedge \xi_i \wedge \bigcirc \psi_i)$$

a disjunction of terms consisting of a timer set operation  $TS_i$  applied to the conjunction of a propositional formula  $\pi_i$ , timer conditions  $\xi_i$  and another  $\text{MITL}_{\leq}$  formula  $\psi_i$  behind a Next operator. When the tableau automaton is constructed, this corresponds to a collection of transitions labelled with the timer set operation  $TS_i$  to locations labelled with propositional symbols matching  $\pi_i$ , timer conditions matching  $\xi_i$  and whose outgoing transitions will be derived from  $\psi_i$ . For instance the formula  $p\mathbf{U}_{\leq 5}q$  has an ( $\stackrel{s}{=}$ ) equivalent formula  $([x := 5].q) \vee ([x := 5].(p \wedge x > 0 \wedge \bigcirc p\mathbf{U}_{\leq x}q))$  in disjunctive temporal normal form.

We mention *some* of the equivalences on which the normal form rewriting procedure is based. The numbers besides the equivalences mention the rules of Table 3, introduced in the next subsection, to which they correspond.

**Instantiating Timers** These equivalences suggest how to use timers to check timing constraints expressed by formulas.

$$\varphi_1 \mathbf{U}_{\leq d} \varphi_2 \stackrel{s}{=} [x := d].(\varphi_1 \mathbf{U}_{\leq x} \varphi_2) \quad (\text{S.1})$$

$$\varphi_1 \mathbf{U}_{\leq d} \varphi_2 \stackrel{o}{=} [x := d].(\varphi_1 \mathbf{U}_{\leq x+\epsilon} \varphi_2) \quad (\text{O.1})$$

**Unfolding** These equivalences are used to separate constraints on the current interval from constraints on the remainder of the state sequence.

$$\varphi_1 \mathbf{U}_{\leq x} \varphi_2 \stackrel{s,o}{=} \varphi_2 \vee (x > 0 \wedge \varphi_1 \wedge \bigcirc \varphi_1 \mathbf{U}_{\leq x} \varphi_2) \quad (\text{C.6})$$

$$\varphi_1 \mathbf{U}_{\leq x+\epsilon} \varphi_2 \stackrel{s,o}{=} \varphi_2 \vee (x \geq 0 \wedge \varphi_1 \wedge \bigcirc \varphi_1 \mathbf{U}_{\leq x+\epsilon} \varphi_2) \quad (\text{C.7})$$

$$\varphi_1 \mathbf{V}_{\leq d} \varphi_2 \stackrel{s}{=} (\varphi_2 \wedge \varphi_1) \vee [y := d].(\varphi_2 \wedge \bigcirc \varphi_1 \mathbf{V}_{\leq y} \varphi_2) \quad (\text{S.2})$$

$$\varphi_1 \mathbf{V}_{\leq d} \varphi_2 \stackrel{o}{=} (\varphi_2 \wedge \varphi_1) \vee (\varphi_2 \wedge \bigcirc \varphi_1 \mathbf{V}_{< d} \varphi_2) \quad (\text{O.2})$$

$$\varphi_1 \mathbf{V}_{< d} \varphi_2 \stackrel{s}{=} (\varphi_2 \wedge \varphi_1) \vee [y := d].(\varphi_2 \wedge \bigcirc \varphi_1 \mathbf{V}_{< y} \varphi_2) \quad (\text{S.3,4})$$

$$\varphi_1 \mathbf{V}_{< y} \varphi_2 \stackrel{s,o}{=} (\varphi_1 \wedge \varphi_2) \vee y \leq 0 \vee (\varphi_2 \wedge \bigcirc \varphi_1 \mathbf{V}_{< y} \varphi_2) \quad (\text{C.8})$$

$$\varphi_1 \mathbf{V}_{\leq y} \varphi_2 \stackrel{s,o}{=} (\varphi_1 \wedge \varphi_2) \vee y < 0 \vee (\varphi_2 \wedge \bigcirc \varphi_1 \mathbf{V}_{\leq y} \varphi_2) \quad (\text{C.9})$$

**Absorbtion** These equivalences enable one to use only a finite number of timers.

$$\varphi_1 \mathbf{U}_{\leq x} \varphi_2 \wedge \varphi_1 \mathbf{U}_{\leq d} \varphi_2 \stackrel{s,o}{=} \varphi_1 \mathbf{U}_{\leq x} \varphi_2 \text{ if } \nu(x) \leq d \quad (\text{C.5})$$

$$\varphi_1 \mathbf{V}_{< d} \varphi_2 \wedge \varphi_1 \mathbf{V}_{\leq y} \varphi_2 \stackrel{s}{=} \varphi_1 \mathbf{V}_{< d} \varphi_2 \text{ if } \nu(y) < d \quad (\text{S.3})$$

$$\varphi_1 \mathbf{V}_{\leq y} \varphi_2 \wedge [y := d].(\varphi_1 \mathbf{V}_{< y} \varphi_2) \stackrel{s}{=} [y := d].\varphi_1 \mathbf{V}_{< y} \varphi_2 \text{ if } \nu(y) < d \quad (\text{S.4})$$

Before we present the normal form procedure, we introduce some more notation to help in the transition from formulas to automata. We identify sets of formulas with the conjunction of the formulas in the set, the empty set being the formula true. We use a triple  $\langle TS, Now, Next \rangle$  to denote the formula  $TS.(Now \wedge \bigcirc Next)$ . A set of such triples is associated with the disjunction of the corresponding formulas.

	CASE $\psi =$	CONDITION	$\Phi \cup \{\langle TS, Now \cup \{\psi\}, Next \rangle\}$ REDUCES TO:
C.1	true		$\Phi \cup \{\langle TS, Now, Next \rangle\}$
C.2	false		$\Phi$
C.3	$\psi_1 \vee \psi_2$		$\Phi \cup \{\langle TS, Now \cup \{\psi_1\}, Next \rangle, \langle TS, Now \cup \{\psi_2\}, Next \rangle\}$
C.4	$\psi_1 \wedge \psi_2$		$\Phi \cup \{\langle TS, Now \cup \{\psi_1, \psi_2\}, Next \rangle\}$
C.5	$\varphi_1 U_{\leq d} \varphi_2$	$\Xi^\psi \cap Now \neq \emptyset$ or $\varphi_2 \in Now$	$\Phi \cup \{\langle TS, Now, Next \rangle\}$
C.6	$\varphi_1 U_{\leq x_\psi} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_2\}, Next \rangle, \langle TS, Now \cup \{x > 0, \varphi_1\}, Next \cup \{\psi\} \rangle\}$
C.7	$\varphi_1 U_{\leq x_\psi + \epsilon} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_2\}, Next \rangle, \langle TS, Now \cup \{x \geq 0, \varphi_1\}, Next \cup \{\psi\} \rangle\}$
C.8	$\varphi_1 V_{< y_\psi} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_1, \varphi_2\}, Next \rangle, \langle TS, Now \cup \{y_\psi \leq 0\}, Next \rangle, \langle TS, Now \cup \{\varphi_2\}, Next \cup \{\psi\} \rangle\}$
C.9	$\varphi_1 V_{\leq y_\psi} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_1, \varphi_2\}, Next \rangle, \langle TS, Now \cup \{y_\psi < 0\}, Next \rangle, \langle TS, Now \cup \{\varphi_2\}, Next \cup \{\psi\} \rangle\}$
S.1	$\varphi_1 U_{\leq d} \varphi_2$	$\Xi^\psi \cap Now = \emptyset$ and $\varphi_2 \notin Now$	$\Phi \cup \{\langle TS[x := d], Now \cup \{\varphi_1 U_{\leq x_\psi} \varphi_2\}, Next \rangle\}$
S.2	$\varphi_1 V_{\leq d} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_2, \varphi_1\}, Next \rangle, \langle TS[y := d], Now \setminus \Xi^\psi \cup \{\varphi_2\}, Next \cup \{\varphi_1 V_{\leq y_\psi} \varphi_2\} \rangle\}$
S.3	$\varphi_1 V_{< d} \varphi_2$	$\varphi_1 V_{\leq y_\psi} \varphi_2 \notin Now$ or $y_\psi := d \in TS$	$\Phi \cup \{\langle TS, Now \cup \{\varphi_2, \varphi_1\}, Next \rangle, \langle TS[y := d], Now \setminus \Xi^\psi \cup \{\varphi_2\}, Next \cup \{\varphi_1 V_{< y_\psi} \varphi_2\} \rangle\}$
S.4		$\varphi_1 V_{\leq y_\psi} \varphi_2 \in Now$ , $y_\psi := d \notin TS$	$\Phi \cup \{\langle TS, Now \cup \{\varphi_2, \varphi_1\}, Next \rangle, \langle TS[y := d], Now \setminus \Xi^\psi \cup \{\varphi_1 V_{< y_\psi} \varphi_2\}, Next \rangle\}$
O.1	$\varphi_1 U_{\leq d} \varphi_2$	$\Xi^\psi \cap Now = \emptyset$ and $\varphi_2 \notin Now$	$\Phi \cup \{\langle TS[x := d], Now \cup \{\varphi_1 U_{\leq x_\psi + \epsilon} \varphi_2\}, Next \rangle\}$
O.2	$\varphi_1 V_{\leq d} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_2, \varphi_1\}, Next \rangle, \langle TS, Now \cup \{\varphi_2\}, Next \cup \{\varphi_1 V_{< d} \varphi_2\} \rangle\}$
O.3	$\varphi_1 V_{< d} \varphi_2$		$\Phi \cup \{\langle TS, Now \cup \{\varphi_2, \varphi_1\}, Next \rangle, \langle TS, Now \cup \{\varphi_2\}, Next \cup \{\varphi_1 V_{< d} \varphi_2\} \rangle\}$

Fig. 3. Rewrite rules for singular (S.n), open (O.n) and common rules (C.n)

## 4.2 The Normal Form Procedure

We now introduce a procedure that rewrites extended MITL<sub>≤</sub> formulas into normal form, and is used as an integral part of the tableau construction.

**Definition 5.** Let  $\Psi$  be a set of formulas, then the normal form  $NF(it, \Psi)$  of  $\Psi$  for an interval of type  $it$  is computed with the following procedure. Let  $P_0 = \{\langle \emptyset, \Psi, \emptyset \rangle\}$ . Then as long as one of the rewrite rules (C.1-9, S.1-4 for  $it = s$  and C.1-9, O.1-3 for  $it = o$ ) of Table 3 applies to any of the terms in  $P_n$ , apply one to a term to obtain  $P_{n+1}$ . The normal form  $NF(it, \Psi)$  is obtained from  $P_n$  when no more rule applies.

It is easy to check that the procedure ends and the resulting formula is in normal form. If no more rule applies, then for every term,  $Now$  contains only propositions and timer conditions. Moreover, the resulting formula is equivalent w.r.t the interpretation  $\models^{it}$ :

**Lemma 1.** Let  $\Psi \subseteq cl(\varphi)$  for some  $\varphi \in \text{MITL}_{\leq}$ ,  $\nu$  be a timer valuation such that  $\nu(x_\psi) \leq d$  for every Until or Release formula  $\psi \in cl(\varphi)$  with bound  $d$ , and  $\bar{p}$  a timed state sequence with interval sequence  $\bar{I}$  that is fine for all basic subformulas

---

```

 $L_0 := \{((s, Now, Next), TS(\bar{0})) \mid \langle TS, Now, Next \rangle \in NF(s, \{\varphi\})\}$ 
 $L_{New} := \{(s, Now, Next) \mid ((s, Now, Next), TS) \in L_0\}$ 
 $L_s := \emptyset, L_o := \emptyset, E := \emptyset$ 
while  $L_{New} \neq \emptyset$  do
  Let  $(it, Now, Next) \in L_{New}$ 
   $L_{New} := L_{New} \setminus \{(it, Now, Next)\}$ 
   $L_{it} := L_{it} \cup \{(it, Now, Next)\}$ 
  for every  $(TS', Now', Next') \in NF(\bar{it}, Next)$  do
     $E := E \cup \{((it, Now, Next), TS', (\bar{it}, Now', Next'))\}$ 
    if  $(\bar{it}, Now', Next') \notin L_s \cup L_o$  then  $L_{New} := L_{New} \cup \{(\bar{it}, Now', Next')\}$ 
  od
od

```

---

**Fig. 4.** Algorithm for constructing the locations and edges of the tableau automaton.

of  $\Psi$ . If  $\bar{\rho} \models_{\nu}^{it} \Psi$ , then there is some term  $\langle it, TS, Now, Next \rangle \in NF(it, \Psi)$  such that  $\bar{\rho} \models_{TS(\nu)}^{it} Now$ , and  $\bar{\rho} \models_{TS(\nu)}^{it} \bigcirc Next$ .

Proofs have been left out due to space limitations. Details can be found in [7].

## 5 Tableau Construction

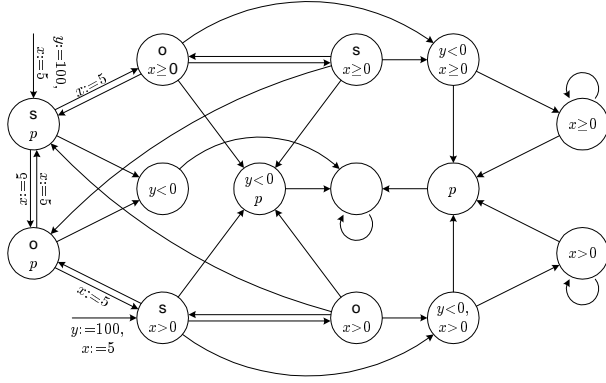
The tableau automaton of an  $\text{MITL}_{\leq}$  formula  $\varphi$  is computed as follows. (With  $it \in \{s, o\}$ , we use  $\bar{it}$  to denote  $o$  if  $it = s$  and  $s$  if  $it = o$ .)

**Definition 6.** Let  $\varphi$  be a basic  $\text{MITL}_{\leq}$  formula and  $Prop$  be the set of atomic propositions that occur in  $\varphi$ . Then the tableau automaton  $A_{\varphi}$  of  $\varphi$  is the automaton  $\langle L_s, L_o, T, L_0, Q, TC, E \rangle$  over the alphabet  $2^{Prop}$ , where

- $T$  is the set of all timers  $x_{\psi}$  for every Until or Release formula  $\psi$  that occurs as a syntactic subformula of  $\varphi$ .
- The locations  $(L_s, L_o)$ , initial extended locations  $(L_0)$  and transitions  $(E)$  are computed by the procedure depicted in Fig. 4. The locations  $\ell \in L_s \cup L_o$  are triples  $(it, Now, Next)$ . The first item of the location  $\ell$  is denoted by  $it(\ell)$ , the second by  $Now(\ell)$  and the last by  $Next(\ell)$ .
- $Q(\ell) = \{\sigma \in 2^{Prop} \mid \forall p \in Prop, p \in Now(\ell) \Rightarrow p \in \sigma, \neg p \in Now(\ell) \Rightarrow p \notin \sigma\}$ . That is, a location  $\ell$  is labelled with all sets of propositions that are consistent with the atomic propositions and the negated atomic propositions in  $Now(\ell)$ .
- $TC(\ell) = TCond(T) \cap Now(\ell)$ , the location is labelled with all timer conditions in  $Now(\ell)$ .

**Example** With a prototype implementation of the algorithm, the following example was produced for the formula  $\Box_{\leq 100} \Diamond_{\leq 5} p = \text{false} \vee_{\leq 100} (\text{true} \cup_{\leq 5} p)$ . We arrive at the automaton represented in Fig. 5. Only the formulas in the *Now* set of the locations have been depicted. The (two) initial extended locations are





**Fig. 5.** Example tableau automaton of the formula  $\Box_{\leq 100} \Diamond_{\leq 5} p$

represented by an arrow not originating from any location leading to the initial location and labelled with a timer setting that yields the initial timer valuation when applied to the timer valuation that assigns 0 to every timer. There is a timer associated with the formula  $\Diamond_{\leq 5} p$  named  $x$  and a timer associated with the formula  $\Box_{\leq 100} \Diamond_{\leq 5} p$  named  $y$ . Singular locations are labelled with ‘s’, open location are labelled with ‘o’. To simplify the figure, locations without ‘s’ or ‘o’ represent two locations, a singular one and an open one, when they are labelled identically (a straightforward optimisation opportunity for the algorithm); without the automaton has 22 locations and 46 edges.

That the presented algorithm is correct, is stated by the following theorem.

**Theorem 1.** *Let  $\varphi$  be a basic MITL $_{\leq}$  formula and  $A_{\varphi}$  the corresponding tableau automaton, then for every timed state sequence  $\bar{\rho}$ ,  $A_{\varphi}$  accepts  $\bar{\rho}$  iff  $\bar{\rho} \models \varphi$ .*

Details and proof can be found in [7].

## 6 Conclusions

We have presented an on-the-fly tableau construction for the fragment MITL $_{\leq}$  of MITL. It employs the introduction of explicit timers and timer set operators into the logic, as well as a Next operator as in [8], but represents a non/trivial improved of [8] by the ability to deal with arbitrary interval sequences, and thereby lifts the restrictions to special intervals and makes it applicable to standard timed systems. In [4] it has been shown that the construction of tableaux for MITL $_{0,\infty}$ , a slightly different fragment of MITL, is PSPACE-complete, and this result can be adapted to the case of MITL $_{\leq}$ . Thus, the theoretical worst-case complexity of our construction is the same as that of the construction in [4]. Being on-the-fly, this algorithm should give much smaller tableaux in practice. Moreover, we have found a small deficiency of the construction of [2,4]. It is believed that it can be resolved with concepts from our algorithm. Extension of the logic with liveness properties, strict bounds and lower bounds instead of upper

bounds are straightforward, but have been left out for clarity. We further think that this work can form the basis for further extensions towards weakly monotonic models of time, often found in interleaving semantics for timed systems and tools such as UPPAAL [11]. Also, optimisations that have been developed for (on-the-fly) tableau constructions in the untimed case (see e.g. [5,6,9]) might be applicable to our algorithm.

## References

1. L. Aceto, A. Burgueño, and K. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Proc. of Tools and Algorithms for Construction and Analysis of Systems, 4th Int. Conf., TACAS '98, Lisbon, Portugal, March 28 - April 4 1998, LNCS Vol. 1384*, pages 263–280, 1998. Springer Verlag.
2. R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.
3. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
4. R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, January 1996.
5. M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In N. Halbwachs and D. Peled, editors, *Proc. of Computer Aided Verification: 11th Int. Conf., CAV'99, Trento, Italy, July 6-10, 1999, LNCS 1633*, pages 249–260. Springer Verlag, 1999.
6. K. Etessami and G. Holzman. Optimising Büchi automata. In C. Palamidessi, editor, *Proc. of the 11th Int. Conf. on Concurrency Theory (CONCUR'2000), Pennsylvania State University, Pennsylvania, USA, August 22-25, 2000, LNCS 1877*, pages 153–167, 2000. Springer Verlag.
7. M.C.W. Geilen. *Formal Techniques for Verification of Complex Real-Time Systems*. PhD thesis, Eindhoven University of Technology, 2002.
8. M.C.W. Geilen and D.R. Dams. An on-the-fly tableau construction for a real-time temporal logic. In M. Joseph, editor, *Proc. of the 6th Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT2000, 20-22 September 2000 Pune, India, LNCS 1926*, pages 276–290, 2000. Springer Verlag.
9. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. IFIP/WG6.1 Symp. Protocol Specification Testing and Verification (PSTV95), Warsaw Poland*, pages 3–18. Chapman & Hall, June 1995.
10. T. Henzinger. It's about time: Real-time logics reviewed. In D. Sangiorgi and R. de Simone, editors, *Proc. of the 9th Int. Conf. on Concurrency Theory (CONCUR'98), Nice, France, 1998, LNCS 1466*, pages 439–454, 1998. Springer Verlag.
11. K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
12. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of 12th Annual ACM Symp. Principles of Programming Languages*, pages 97–107. ACM SIGACT/SIGPLAN, 1985.
13. A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Ann. Symp. on Found. of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.
14. M.Y. Vardi and P. Wolper. An Automata-Theoretic approach to automatic program verification. In *Proc. of Logic in Computing Science*, 1986.

# Strengthening Invariants by Symbolic Consistency Testing\*

Husam Abu-Haimed, Sergey Berezin, and David L. Dill

Stanford University  
{husam,berezin,dill}@stanford.edu

**Abstract.** We describe a relatively simple method for strengthening invariants when verifying infinite-state hardware systems called *symbolic consistency testing*. The method requires a high-level symbolic simulator and a decision procedure for quantifier-free first-order logic. The user only needs to provide a small number of simple symbolic test vectors that expose internal inconsistencies in the system state. A verification system then uses symbolic simulation with these test vectors to strengthen the original invariant to an inductive one, which is discharged using  $k$ -step induction. The main advantage of our method is that the user input is usually very simple and intuitive, and the user does not need to be exposed to the actual complexity of the verification, which then proceeds completely automatically. The method is applied to several typical microarchitectures for cached memories.

## 1 Introduction

There are many potential advantages to verifying high-level descriptions of hardware systems, before commitments to implementation details such as data-path width are made. Errors that are best corrected by major design or architecture changes can be detected while there is still time to fix them properly, rather than patching a fundamentally-flawed design when it is too late to do anything else. Also, the verification effort is more valuable since results can be re-used for many different ways of implementing the design.

Unfortunately, most of the current practical formal verification methods, such as model checking, cannot be directly applied before implementation details are fixed, because they cannot deal with systems with unbounded numbers of states. The most widely used tools for formal verification of high-level system descriptions are interactive theorem provers, such as HOL or PVS [7,12], or even manual proof. However, even with steady improvement in interactive theorem provers, proofs can be very tedious.

---

\* This research was supported by GSRC contract DABT63-96-C-0097-P00005, by National Science Foundation CCR-0121403, and by King Fahd University of Petroleum and Minerals, Saudi Arabia. The content of this paper does not necessarily reflect the position or the policy of GSRC, NSF, or the Government, and no official endorsement should be inferred.

We describe a methodology, called *symbolic consistency testing*, that addresses one of the core problems in verifying non-trivial systems: computing inductive invariants. Finding invariants is a major problem even for most realistic finite-state systems; for relatively small finite-state systems the problem can be automatically solved by model checkers. Although there is no general-purpose algorithm to find inductive invariants automatically for infinite-state systems, it is possible to accumulate an inventory of incomplete, usually domain-specific techniques to help find them.

Symbolic consistency testing exploits symbolic simulation and a decision procedure for a quantifier-free fragment of first-order logic. The prototype implementation of the idea is based on the CVC tool [13], a general-purpose decision procedure for quantifier-free first-order logic, but the same ideas could be used with an interactive theorem prover if it has adequate support for efficient symbolic simulation and can effectively solve the logic formulas in question.

The user input for this verification method is the design to be verified, an invariant property to be proved (which is not inductive, in general), and a symbolic consistency test consisting of several short symbolic test vectors that will expose inconsistencies in the internal state of the system. The symbolic test vectors are symbolically simulated by the verification system to produce logical terms, which are then used to strengthen the original invariant to make it inductive. The resulting invariant may be very large and complex, but most of it is constructed automatically, so the user does not need to see it. This new invariant is then proven by a  $k$ -step induction (for some fixed  $k$  that depends on the design).

We demonstrate the use of the technique to verify the microarchitectures of *cached memory systems*. No properties are assumed about the addresses and values of the memory system, so any proved properties will hold for a large number of possible implementations of the microarchitecture. For these systems, the symbolic consistency tests involve reading a small number of distinct symbolic addresses.

The verification problem is to compare a memory system microarchitecture with an idealized view of the memory as a simple uncached array. The microarchitecture and the array can be regarded as a single system consisting of the two subsystems running in parallel with identical inputs. Given executable descriptions of the memory system microarchitecture and the array, the property can be formalized as an invariant, which is usually not inductive, and requires some consistency condition on this combined system for every state that is reachable from some given initial state.

The resulting predicate describes the behavior of the system over several cycles of execution. Therefore, it is helpful to extend the basic method of proving invariants to span several cycles as well, so invariant resulting from consistency testing is proved using *k-step induction*, where  $k$  is some constant that depends on the design.

Many techniques have been developed in the literature for finding invariants automatically [9,14,16,2,1,11]. In spite of the sophistication of these techniques,

the process of finding invariants is still mostly manual. Symbolic simulation has been used as a tool to reduce the manual effort in constructing the invariants [14,15,17].

Manna and Pnueli [9] showed methods based on bottom-up techniques for verifying temporal properties of reactive systems, which are then used to extract assertions implied by the transition relation. These assertions are used to strengthen the invariant.

Su *et al.* [14] presented some heuristics that automatically generate some of the invariants needed in processor verification based on syntactic manipulations of the transition function of the design. These techniques are classified as bottom-up methods.

Among the most successful approaches to proving invariants and other properties for infinite-state systems are those based on predicate abstraction [3,6,8,4,5]. In predicate abstraction, the infinite-state system is abstracted into a finite system using a set of predicates on the system variables. These predicates are usually provided by the user. If the properties of interest hold for the finite abstract system, they hold for the infinite system.

Another class of techniques called top-down methods start with the property to be proved and use backward propagation with different heuristics to strengthen that property. Combinations of the top-downs and bottom-up techniques [16,2] have also been developed.

## 2 Induction on Time

We model a hardware design as a *transition system*

$$T = (S, s_0, N, R, D_{\text{in}}, D_{\text{out}}),$$

where  $S$  is a non-empty (and possibly infinite) set of states,  $s_0 \in S$  is the *initial state*,  $D_{\text{in}}$  and  $D_{\text{out}}$  are the domains of *inputs* and *outputs*,  $N : S \times D_{\text{in}} \rightarrow S$  is the *transition function*, and  $R : S \times D_{\text{in}} \rightarrow D_{\text{out}}$  is the *output function*. Intuitively, a *run* of a transition system on an input sequence  $\alpha^\ell \in D_{\text{in}}^*$  of length  $\ell$  is a sequence of states  $\pi = s_0 s_1 \dots s_\ell$  starting from the initial state  $s_0$ , such that  $s_{i+1} = N(s_i, \alpha_i)$  for all  $0 \leq i \leq \ell - 1$ , producing a sequence of outputs  $\xi^\ell \in D_{\text{out}}^*$ , where each  $\xi_i = R(s_i, \alpha_i)$ . We write  $N(s, \alpha^\ell)$  to denote the final state of running  $T$  on the input sequence  $\alpha^\ell$  starting from the state  $s$ :

$$N(s, \alpha^\ell) = N(\underbrace{N(\dots N(s, \alpha_0), \alpha_1), \dots, \alpha_{\ell-1}}_\ell).$$

In particular,  $N(s, \alpha^\ell) = s$  when  $\ell = 0$  (the input sequence is empty). It is important to note that a single transition in  $T$  can actually represent a complex transaction in the real hardware implementation requiring multiple cycles of execution. Therefore, the next state function  $N$  may have a quite complex definition.

A state  $s$  is called *reachable* in a transition system  $T$ , if there is an input sequence  $\alpha^\ell$  such that  $s = N(s_0, \alpha^\ell)$ , where  $s_0$  is the initial state of  $T$ . In this paper, we only consider *safety properties*, or *invariants* over the set of reachable states. We say that a transition system  $T$  satisfies a safety property  $Q(s)$ , if  $Q(s)$  holds for every reachable state  $s$  of  $T$ :

$$\forall s. \text{reachable}(s) \Rightarrow Q(s). \quad (1)$$

This can be equivalently rewritten as follows:

$$\forall \ell, \alpha^\ell. Q(N(s_0, \alpha^\ell)). \quad (2)$$

The conventional way of proving (2) is by induction on time, when  $Q$  is first shown to hold in the initial state  $s_0$ , and then the transition function  $N$  is shown to preserve  $Q$ :

$$\begin{aligned} &Q(s_0) \\ &\forall s, \sigma. Q(s) \Rightarrow Q(N(s, \sigma)). \end{aligned} \quad (3)$$

This simple induction, however, fails most of the time in practice, because the property we want to prove is not inductive and needs to be strengthened. Often, the problem of finding an inductive invariant is the hardest part of verification, which requires extensive manual guidance and depends greatly on the class of problems to be solved.

In the remainder of this section we introduce a new technique for strengthening an important class of invariants called *functional equivalence* and justify its soundness. The technique gives the user a simple recipe on how to collect the additional information about the design, and the rest of the verification proceeds completely automatically.

## 2.1 Functional Equivalence

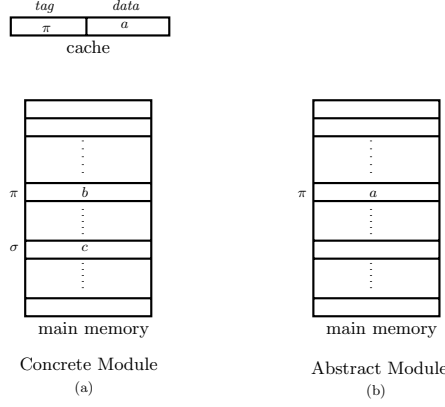
We prove correctness of systems using the idea of *functional equivalence*. The problem is stated as follows. Given two systems, the concrete system  $T^c$  (the system we want to verify) and the abstract system  $T^a$  (which defines the required functionality of  $T^c$ ), prove that  $T^c$  is functionally equivalent to  $T^a$ . Two systems are said to be functionally equivalent if they produce the same sequence of outputs for the same sequence of inputs. Formally, this is expressed as follows.

The states  $s^c$  and  $s^a$  of  $T^c$  and  $T^a$  respectively are said to be functionally equivalent if both systems produce the same output for any given input in those states:

$$\forall \lambda. R^c(s^c, \lambda) = R^a(s^a, \lambda). \quad (4)$$

The entire system  $T^c$  is said to be functionally equivalent to  $T^a$  if any reachable state  $s^c$  of  $T^c$  reached by some input sequence  $\alpha^\ell$  is functionally equivalent to the corresponding state  $s^a$  of  $T^a$  reached by the same input sequence:

$$\forall \ell, \alpha^\ell, \lambda. R^c(N^c(s_0^c, \alpha^\ell), \lambda) = R^a(N^a(s_0^a, \alpha^\ell), \lambda). \quad (5)$$

**Fig. 1.** Memory Example

Effectively, we construct a *product transition system*

$$T = (S^c \times S^a, (s_0^c, s_0^a), N, R, D_{\text{in}}, D_{\text{out}}),$$

where

$$\begin{aligned} N((s^c, s^a), \alpha) &= (N^c(s^c, \alpha), N^a(s^a, \alpha)) \\ R((s^c, s^a), \alpha) &= (R^c(s^c, \alpha), R^a(s^a, \alpha)) \end{aligned}$$

and define  $Q(s)$  to be  $\forall \lambda. R^a(s^a, \lambda) = R^c(s^c, \lambda)$ . The functional equivalence property (5) then becomes  $\forall \ell, \alpha^\ell. Q(N((s_0^c, s_0^a), \alpha^\ell))$ , which is the same as formula (2). So, we can use the same induction principle given by (3) to prove the functional equivalence (5) of the two modules.

## 2.2 Example: One-Line Cache

Consider a small example of a read-only memory with a single-line cache given in figure 1(a). To verify the correctness of this design, we show that it is functionally equivalent to a simple (uncached) array of data in figure 1(b). Since the memories are read-only, the input to both modules is the address, and the output is the data read from that address:

$$\begin{aligned} D_{\text{in}} &= \text{Addr} \\ D_{\text{out}} &= \text{Data}. \end{aligned}$$

The transition systems  $T^c$  and  $T^a$  are defined as follows. The abstract state  $s^a$  of  $T^a$  is just an array  $M$  indexed by  $\text{Addr}$  and holding the **Data** elements. The next state function  $N^a$  is the identity function, and  $R^a(s^a, \lambda) = M[\lambda]$ . The concrete state  $s^c$  of  $T^c$  contains the state of the cache in addition to the same array  $M$ . Initially, in  $s_0^c$ , the cache is empty. The next state function  $N^c(s^c, \lambda)$  adds the address  $\lambda$  and the data stored under that address  $M[\lambda]$  to the cache,

yielding the new state. The output function  $R^c(s^c, \lambda)$  is similar to  $N^c$ , except that it returns the data associated with the address  $\lambda$ .

Unfortunately, proving the functional equivalence of the two memories by simple induction fails. Consider a state in which the cache in the concrete memory contains the data  $a$  for an address  $\pi$ , but the main memory has a different value  $b$  under the same address. The two memories  $T^c$  and  $T^a$  are functionally equivalent in the current state  $s$  if the abstract memory happens to contain  $a$  for the address  $\pi$ , thus, the induction hypothesis  $Q(s)$  is satisfied. However, transitioning to the next state by reading some address  $\sigma \neq \pi$  brings  $T^c$  to a new state  $s'$  where the address  $\pi$  is no longer cached. Therefore, reading  $\pi$  again yields  $b \neq a$  (since it has to come from the main memory), which no longer agrees with  $T^a$ .

The induction fails in this case because it starts out from an inconsistent state, which is not reachable. The natural way to strengthen the invariant is to require the state to be *consistent*. In this example it means that the cached value must be the same as in the main memory (this property is called *coherence*).

Fortunately, there is a very simple way to test for consistency. The idea is to read from the cached address  $\pi$ , then from some other address, to make sure that the original value in the cache is replaced, then read from  $\pi$  again, and require that the results of the first and the last reads are the same. The second read from  $\pi$  forces the cache to reload the value from the main memory, hence, this effectively requires the originally cached value to be consistent with the main memory.

This consistency test is very intuitive, and can be easily constructed by the designer. In our approach, supplying this information is sufficient to strengthen the invariant and finish the verification completely automatically. So, the user can use the tool as a black box, without having to know the details of the algorithm, or be an expert in formal verification.

Notice that the consistency test spans several consecutive states of execution, which requires a more general *k-step induction*, which we define formally in the next section.

### 2.3 k-Step Induction

Formally, in order to show  $\forall \ell, \alpha^\ell. Q(N(s_0, \alpha^\ell))$ , it is sufficient to prove two formulas representing the *base case* and the *induction step*:

$$\forall \ell < k \quad \forall \alpha^\ell. Q(N(s_0, \alpha^\ell)) \quad (6)$$

$$\forall s, \sigma^k. \left[ \bigwedge_{\ell=0}^{k-1} \forall \alpha^\ell. Q(N(s, \alpha^\ell)) \right] \Rightarrow Q(N(s, \sigma^k)) \quad (7)$$

In practice, it is possible to prove a stronger and a slightly simpler formula for the induction step (7):

$$\forall s, \sigma^k. \exists \alpha^{k-1}. \left[ \bigwedge_{\ell=0}^{k-1} Q(N(s, \alpha^\ell)) \right] \Rightarrow Q(N(s, \sigma^k)), \quad (8)$$



where  $\alpha^\ell$  is a prefix of  $\alpha^{k-1}$  for all  $\ell < k$ . So, if there is a  $k$  such that (6) and (8) hold, we conclude that (2) is true.

Note that formula (8) contains an existential quantifier, and since solving validity of existential first-order formulas is undecidable, some manual guidance may be necessary to eliminate the existential quantifiers. In section 3.1, we propose a new instantiation method which significantly reduces the required manual effort. Decidability is not an issue for the base case of the induction (6), since the universal fragment of the logic is decidable and is directly supported by CVC, our validity checker.

## 2.4 Symbolic Simulation

Intuitively, a symbolic simulation of a transition system  $T$  is just like an ordinary simulation, only the input sequence  $\alpha^\ell$  and the starting state  $s$  are symbolic variables rather than constant values. Therefore, the resulting run over  $\alpha^\ell$  becomes the sequence of symbolic terms representing the states of the run:

$$s, N(s, \alpha^1), N(s, \alpha^2), \dots, N(s, \alpha^\ell), \quad (9)$$

where each  $\alpha^i$  is a prefix of  $\alpha^\ell$ . One can think of such a symbolic run as “the most general run” of  $T$  of length  $\ell$ , since any other run of the same length can be obtained from (9) by instantiating  $\alpha^\ell$  with concrete values.

For practical examples, the next state and output functions  $N(s, \alpha)$  and  $R(s, \alpha)$  are explicitly defined as terms over  $s$  and  $\alpha$ , and each symbolic state becomes a complex term. Moreover, real hardware circuits often require multiple cycles to complete a single operation, which in  $T$  is treated as “one step” of execution. So, in reality,  $N(s, \alpha)$  may denote the final state of a long sequence of intermediate, *unobservable* states of the real implementation. This fact significantly contributes to the size of the resulting state terms. The task of a symbolic simulator is to build the state terms efficiently, which in practice may be as large as hundreds of thousands of nodes.

The importance of symbolic simulation to formal verification is in the following observation. If a safety property  $Q$  is proven for a symbolic run of length  $\ell$ , then  $Q$  holds for any concrete run of length  $\ell$  of  $T$ . This provides a method for proving formulas (6) and (8). Obviously, proving  $Q(s)$  where  $s$  is a huge term is a challenging task, and a powerful theorem prover or a *validity checker* engine is required. In this work, CVC is used both as a symbolic simulator and a validity checker.

## 3 Soundness of Consistency Testing

In section 2.2, we observed that  $k$ -step induction captures the consistency assumption that is necessary for the induction step to go through. Specifically, for the example in that section, the inductive step in the 2-step induction can be proven valid. From our experience, for virtually any memory design there exists

some  $k$  such that the correctness property for the design can be proven by  $k$ -step induction. The only remaining problem in the induction step (8) is that it contains existential quantifiers:

$$\forall s, \sigma^k \exists \alpha^{k-1}. \left[ \bigwedge_{\ell=0}^{k-1} Q(N(s, \alpha^\ell)) \right] \Rightarrow Q(N(s, \sigma^k)).$$

The existential quantifiers need to be eliminated before the formula can be handed to the validity checker. In general, finding an instantiation for existential quantifiers is undecidable, and some manual guidance may be required. In our approach, we try to minimize the amount of user guidance and reduce it to expressing just the essence of the consistency property needed for the proof. For instance, in section 2.2 the idea behind the consistency test was to read from two distinct addresses to flush the cache and bring the system to a consistent state.

It is often the case that the important part of an instantiation for an existential formula is not a concrete term, but a certain property that the instantiation has to satisfy. Therefore, it is much easier and more intuitive to supply the property of importance explicitly and let the tool fill in the details. This idea can be formalized as a technique called *predicate instantiation*.

### 3.1 Predicate Instantiation of Existential Quantifiers

The existential quantifier in a formula of the form:

$$\forall x. \exists y. \Psi(x, y) \tag{10}$$

is usually eliminated by constructing a function  $f(x)$  and replacing  $y$  for  $f(x)$ :

$$\forall x. \Psi(x, f(x)).$$

However, constructing such a function explicitly for memory examples is a very tedious process. Instead, we apply a different technique which we call *predicate instantiation*. The idea is to find a predicate or a formula  $\Phi(x, y)$  such that the following two formulas can be proven valid:

$$\forall x. \exists y. \Phi(x, y) \tag{11}$$

$$\forall x. \forall y. \Phi(x, y) \Rightarrow \Psi(x, y). \tag{12}$$

It is not hard to derive that the validity of (11) and (12) imply the validity of (10).

Let  $P(s, \lambda)$  stand for  $R^c(s^c, \lambda) = R^a(s^a, \lambda)$ . When applying  $k$ -step induction to the example in section 2.2 (where  $k = 2$ ), the inductive step (8) becomes

$$\forall s. \forall \sigma^2. \exists \alpha. [\forall \beta_0. P(s, \beta_0) \wedge \forall \beta_1. P(N(s, \alpha), \beta_1)] \Rightarrow \forall \lambda. P(N(s, \sigma^2), \lambda).$$

Pulling all the quantifiers to the top level yields the following:

$$\forall s. \forall \sigma^2. \forall \lambda. \exists \alpha. \exists \beta^2. [P(s, \beta_0) \wedge P(N(s, \alpha), \beta_1)] \Rightarrow P(N(s, \sigma^2), \lambda). \tag{13}$$

Recall that in order to test for consistency, we must read from the address  $\lambda$  to test if  $P(s, \lambda)$  is true, then read from some other address  $\alpha$  to get  $N(s, \alpha)$ , and then read from  $\lambda$  again to test if  $P(N(s, \alpha), \lambda)$  is true. If  $P(s, \lambda)$  and  $P(N(s, \alpha), \lambda)$  are true, then the memory system is consistent for this example. These requirements can be expressed by the following formula:

$$\Phi : \beta_0 = \beta_1 = \lambda \wedge \alpha \neq \lambda, \quad (14)$$

which we use for predicate instantiation of existential quantifiers in (13):

$$\forall s \forall \sigma^2 \forall \lambda. \exists \alpha \exists \beta^2. \Phi \quad (15)$$

$$\forall s \forall \sigma^2 \forall \lambda. \forall \alpha \forall \beta^2. [\Phi \wedge P(s, \beta_0) \wedge P(N(s, \alpha), \beta_1)] \Rightarrow P(N(s, \sigma^2), \lambda). \quad (16)$$

The formula (15) is trivially true if the address space  $D_{\text{in}}$  has more than one address, and (16) is shown to be valid by CVC, our automatic validity checker.

Note that to use the instantiation predicate, one still needs to prove an existential formula (11), which is undecidable in general. However, for all of the memory examples that we have tried, the resulting formula was simple enough to be solved automatically by a theorem prover. The theorem prover we used was Otter [10] which took a fraction of a second to prove each of these formulas.

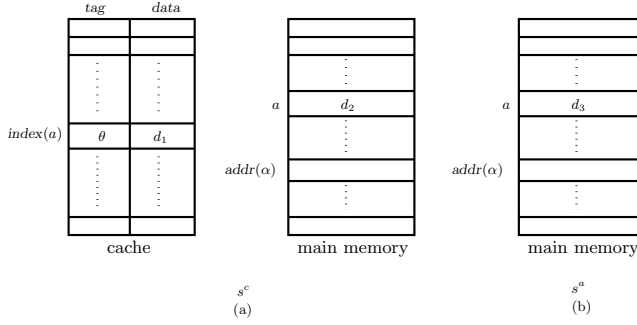
## 4 Examples

To illustrate the use of the methodology described above on practical examples, we applied it to several memory designs with different cache architectures. The write-through direct mapped cache with a parameterized cache size is discussed in detail in section 4.1, and its extension to the multi-level cache design is given in section 4.2.

As we mentioned earlier, the base case of the inductive proof falls into a decidable fragment of the first-order logic, and does not present a problem in verification. Therefore, in the following examples, we concentrate only on proving the induction step.

### 4.1 Write-through Direct Mapped Cache

The memory design given in figure 2(a) is a generalization of the simple example in section 2.2 to a read/write memory with a write-through direct mapped cache and an arbitrary number of cache lines. The figure describes only the high level organization of the system. There are several intermediate registers and buffers in the system that we do not show to simplify the discussion. The system also has a state machine that consists of 6 states controlling the cache in addition to a 3-state state machine that controls the main memory. Simulating a read or a write transaction in this system can take up to 10 cycles. An invariant for such a system needs to describe the values of the cache, the main memory, the intermediate registers, and the state machines at the different cycles of the transaction. That can be a very complex invariant. In this section we show how

**Fig. 2.** Direct Mapped Cache

a simple symbolic consistency test for this system helps us avoid this complexity. The consistency test we show was enough for the induction step of the proof to go through.

The inputs to the memory consist of the read/write flag, the address, and the data in the case of the write flag. On the read input, the memory returns the requested data as an output, and write produces no output, which we denote by a special symbol  $\perp$ . Formally, we define the input and output domains of the transition system  $T^c$  representing the design as follows:

$$D_{in} = (\{\text{read}\} \times \text{Addr}) \cup (\{\text{write}\} \times \text{Addr} \times \text{Data})$$

$$D_{out} = \text{Data} \cup \{\perp\}$$

As in section 2.1, we define the correctness of  $T^c$  to be the functional equivalence between  $T^c$  and the reference model  $T^a$  (figure 2(b)), which is simply an array of data. We refer to  $T$  as the product of  $T^c$  and  $T^a$ ; so a state  $s$  of  $T$  is a pair of states  $(s^c, s^a)$  of the two systems.

For an input  $\lambda = \langle \text{rw}, a, [d] \rangle$ , we write  $\text{rw}(\lambda)$  and  $\text{addr}(\lambda)$  to denote the read/write flag and the address parts of  $\lambda$  respectively. Each address  $a$  consists of two components:  $\text{index}(a)$  and  $\text{tag}(a)$ . The index part is used to identify the cache line (so, the cache array is indexed by  $\text{index}(a)$ ), and the tag is stored in the cache along with the data to reconstruct the complete address.

Following our methodology, a consistency test needs to be constructed for  $T^c$ , which in this case means that the data stored in every cache line is coherent with the main memory. Similarly to the example in section 2.2, consistency of each cache line can be checked by first reading the cached address  $a = \text{addr}(\lambda)$ , then some other address  $a' = \text{addr}(\alpha)$  with the same index as  $a$  to replace the data in that cache line, then read from  $a$  again and compare the result with the first read. This intuitive description is translated into the following formula:

$$\Phi \equiv \beta_0 = \beta_1 = \lambda$$

$$\wedge \text{index}(\text{addr}(\alpha)) = \text{index}(\text{addr}(\lambda)) \wedge \text{tag}(\text{addr}(\alpha)) \neq \text{tag}(\text{addr}(\lambda)).$$

This is all the user needs to supply to the tool, and the rest is done completely automatically.

Since three reads are required by the consistency test, two of which are consecutive (from  $\alpha$  and  $\beta_1$ ), 2-step induction is used. The formula (13), which is the form of the induction step we need to prove, becomes

$$\forall s, \sigma^2, \lambda, \exists \alpha, \beta^2. [P(s, \beta_0) \wedge P(N(s, \alpha), \beta_1)] \Rightarrow P(N(s, \sigma^2), \lambda), \quad (17)$$

where  $P(s, \lambda) \equiv R^c(s^c, \lambda) = R^a(s^a, \lambda)$ .

Observe, that when  $\lambda = \langle \text{write}, a, d \rangle$ , formula (17) is trivially true, since  $R^c(N^a(s^a, \sigma^2), \lambda) = R^a(N^a(s^a, \sigma^2), \lambda) = \perp$ . Therefore, we only need to consider the case when  $\lambda = \langle \text{read}, a \rangle$ .

Instantiating (17) with the predicate  $\Phi$  above yields the following:

$$\forall s, \sigma^2, \lambda, \exists \alpha, \beta^2. \Phi \quad (18)$$

$$\forall s, \sigma^2, \lambda, \forall \alpha, \beta^2. [\Phi \wedge P(s, \beta_0) \wedge P(N(s, \alpha), \beta_1)] \Rightarrow P(N(s, \sigma^2), \lambda). \quad (19)$$

Formula (18) is valid whenever there exists more than one tag for the same index (which is the case for any real design), and (19) is proven valid by our validity checker CVC.

Recall, that the next state function in this example requires 10 cycles of symbolic simulation to compute. Therefore, the formula for the 2-step induction involves 20 cycles of symbolic simulation total. Moreover, due to the intermediate buffers, simply requiring that the cached value is consistent with the main memory is not enough for the invariant to become inductive.

## 4.2 Two-Level Write-through Direct-Mapped Cache

The system we deal with here has two levels of cache, each one of these two levels has an organization similar to that in the previous section. A transaction in this system can take up to 17 cycles. For this example, two levels of cache flushing are required for the consistency test, and therefore, we need 3-step induction. First, we find an address  $\alpha_0 \neq \lambda$  which maps to the same line of the L1 cache:  $\text{index}_1(\alpha_0) = \text{index}_1(\lambda)$ , and read from  $\lambda$ , then  $\alpha_0$ , then  $\lambda$  again. This ensures that if  $\lambda$  is cached both in L1 and L2, the cached data is the same in both caches. Next, we find another address  $\alpha_1 \neq \lambda$  such that  $\alpha_1$  and  $\lambda$  map to the same cache lines *both* in L1 and L2:

$$\text{index}_1(\alpha_1) = \text{index}_1(\lambda) \wedge \text{index}_2(\alpha_1) = \text{index}_2(\lambda).$$

Reading from  $\lambda$ , then  $\alpha_1$ , then  $\lambda$  again clears both caches at the same time and forces the data under the address  $\lambda$  to be fetched from the main memory. Putting it all together yields the following instantiation predicate:

$$\begin{aligned} \Phi \equiv & \beta_0 = \beta_1 = \beta_2 = \lambda \wedge \lambda \neq \alpha_0 \neq \alpha_1 \neq \lambda \wedge \text{index}_1(\alpha_0) = \text{index}_1(\lambda) \\ & \wedge \text{index}_1(\alpha_1) = \text{index}_1(\lambda) \wedge \text{index}_2(\alpha_1) = \text{index}_2(\lambda). \end{aligned} \quad (20)$$

Notice, that  $\Phi$  is an acceptable predicate (i.e. it satisfies (11)) only under the assumption that any cache line has several distinct addresses that map to it

(three for L1 and two for L2), and, moreover, that for any address  $\lambda$  there is another address  $\alpha_1$  which maps to exactly the same cache lines as  $\lambda$  throughout all the cache levels. Both assumptions are quite reasonable for real designs.

As before, the next state function  $N$  for this example requires 17 cycles of symbolic execution, yielding 51 cycles total for the 3-step induction. Again, the actual inductive invariant requires much more information about the intermediate buffers and registers than the high-level description above provides.

This consistency test can be generalized to memory designs with  $n$ -level caches for any concrete  $n$ , and verified using  $(n + 1)$ -step induction.

We applied our approach to many other examples with different cache organizations such as two-way set-associative caches, and write-back caches. In all cases, the instantiation predicate was very compact and easy to construct, and no other manual guidance was required to complete the verification.

One exception is the two-way set-associative cache, where an additional modest manual intervention reduced the original 5-step induction to 3 steps in order to speed up the validity checking. The additional property is that the same address can be cached in at most one cache line.

## 5 Conclusion

This paper addresses one of the most difficult problems in verifying safety properties for infinite-state systems: finding inductive invariants. This problem is undecidable in general and often requires substantial ingenuity and manual guidance from the user.

The proposed methodology reduces the user's burden to understanding the bare essence of the inductive invariant and constructing a relatively simple predicate to help the otherwise automatic tool to finish the proof of the invariant.

Although much simpler than finding an inductive invariant manually, constructing the instantiation predicate can still be a challenging task. However, for cached memories, which is an important class of hardware designs, there is a practical strategy which greatly helps the user to construct such a predicate. As we have observed, the main reason why the invariant is inductive is because any reachable state is *consistent* with itself; that is, the cached data must always be consistent with the main memory. Such a consistency test is often intuitive and easy to construct for a person familiar with the design, and is sufficient to complete the verification automatically. This methodology is actually not restricted to only cached memories, and is applicable to many other classes of designs.

The approach is still in its development stage, and a lot remains to be done. One obvious problem is that the formulas resulting from symbolic simulation are often enormous, which requires a powerful validity checker. Much research needs to be done to optimize the validity checker to the class of formulas that arise in this approach.

Another direction of research is to find heuristics to help the user find the instantiation predicates automatically, and to expand the class of designs this approach can handle.

**Acknowledgements.** The authors would like to thank Aaron Stump, Clark Barrett and Satyaki Das for their help and support with the verification tools, William McCune for his help with Otter, and the anonymous reviewers for the insightful comments on the paper.

## References

1. Saddek Bensalem, Yassine Lakhnech, and Hassen Saidi. Powerful techniques for the automatic generation of invariants. In *Computer Aided Verification*, 1996.
2. Nikolaj Björner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. In *Theoretical Computer Science*, 1997.
3. Michael Colon and Tomas E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, volume 1427 of *LNCS*, pages 293–304. Springer Verlag, 1998.
4. Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, 2001. June 2001, Boston, USA.
5. Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002.
6. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
7. *The HOL System Tutorial*, 1994. <http://www.cl.cam.ac.uk/Research/HVG/HOL/>.
8. D. Lesens and H. Saidi. Automatic verification of parameterized networks of processes by abstraction. In *the 2nd International Workshop on the Verification of Infinite State Systems (INFINITY'97)*, July 1997.
9. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1993.
10. William W. McCune. Otter 3.0 reference manual and guide. Technical report, ANL, 1994.
11. John Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In *5th SPIN workshop*, 1999.
12. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.
13. A. Stump, C. Barrett, and D. Dill. CVC: a Cooperating Validity Checker. In *14th International Conference on Computer-Aided Verification*, 2002.
14. Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic generation of invariants in processor verification. In *FMCAD*, 1996.
15. Jeffrey X. Su, David L. Dill, and Jens U. Skakkebak. Formally verifying data and control with weak reachability invariants. In *FMCAD*, 1998.
16. A. Tiwari, H. Rueß, H. Saidi, and N. Shankar. A technique for invariant generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2001.
17. Miroslav N. Velev and Randal E. Bryant. Formal verification of superscalar microprocessors with multicycle functional units, exceptions, and branch prediction. In *Design Automation Conference (DAC)*, 2000.

# Linear Invariant Generation Using Non-linear Constraint Solving

Michael A. Colón, Sriram Sankaranarayanan, and Henny B. Sipma \*

Computer Science Department  
Stanford University  
Stanford, CA 94305-9045  
{colon,srirams,sipma}@cs.stanford.edu

**Abstract.** We present a new method for the generation of linear invariants which reduces the problem to a non-linear constraint solving problem. Our method, based on *Farkas' Lemma*, synthesizes linear invariants by extracting non-linear constraints on the coefficients of a target invariant from a program. These constraints guarantee that the linear invariant is inductive. We then apply existing techniques, including specialized quantifier elimination methods over the reals, to solve these non-linear constraints. Our method has the advantage of being complete for inductive invariants. To our knowledge, this is the first sound and complete technique for generating inductive invariants of this form. We illustrate the practicality of our method on several examples, including cases in which traditional methods based on abstract interpretation with widening fail to generate sufficiently strong invariants.

## 1 Introduction

An *invariant* assertion of a program at a location is an assertion over the program variables that remains true whenever the location is reached. Invariants are essential for verifying the correctness of programs. The automatic generation of invariants is useful both as a direct means of checking program specifications and as an indirect means of obtaining *intermediate assertions* that can be used as lemmas for proving other safety and liveness properties [16].

An assertion is said to be *inductive* at a program location if it holds the first time the location is reached and is preserved under every cycle back to the location. It has been established that all inductive assertions are invariant. Furthermore, the standard method for proving a given assertion invariant is to find an inductive assertion that strengthens it [16]. Thus invariant generation methods are, normally, methods for generating inductive assertions.

The dominant invariant generation technique is *abstract interpretation* [6]. The main idea behind this approach is to perform an approximate symbolic

---

\* This research was supported in part by NSF(ITR) grant CCR-01-21403, by NSF grant CCR-99-00984-001, by ARO grant DAAD19-01-1-0723, and by ARPA/AF contracts F33615-00-C-1693 and F33615-99-C-3014.



execution of the program until an assertion is reached that remains unchanged by further executions of the program. This assertion can be shown to be inductive, and hence invariant. However, in order to guarantee termination, the method introduces imprecision by use of an extrapolation operator called *widening*. This operator often causes the technique to produce weak invariants. The design of a widening operator with some guarantee of completeness remains a key challenge for abstract interpretation based techniques [7,1]. In fact, tools like HyTECH [12] have given up widening in favor of extrapolation heuristics with no convergence guarantees and have reported good results.

In this paper, we generate linear invariants for linear transition systems, a class of programs that is widely studied [7,1,11]. A large number of reactive systems may be modeled directly or approximately as linear transition systems [13]. Rather than perform a least fixed point computation by iteration, we solve constraints on the coefficients  $c_1, \dots, c_n, d$  of a target invariant  $c_1x_1 + \dots + c_nx_n + d \leq 0$ . The constraints encode the conditions for inductiveness of the inequality. The solution method is exact, thus guaranteeing that all inductive invariants of this form can be found. The main advantage is that there is no heuristic widening operation, and thus the method does not suffer from the problem of overshooting invariants. On the other hand, the method generates *non-linear* constraints that can be solved only for small or medium size problem instances using current techniques. This disadvantage notwithstanding, we demonstrate our approach on some examples drawn from the literature. We show that in many cases, our method generates invariants that forward propagation with widening misses. Non-linear constraint solving is an active area of research, and our approach will become increasingly practical as more effective techniques for solving these constraints are developed.

The rest of the paper is organized as follows: Section 2 presents some preliminary definitions. In Section 3, we present a method for generating constraints using Farkas' Lemma. Techniques for solving non-linear constraints are briefly described in Section 4. Section 5 illustrates the method on several examples, and finally, Section 6 concludes with a discussion of the advantages and drawbacks of the approach.

## 2 Preliminaries

In this section, we provide some key definitions and present Farkas' Lemma, the basis of our approach.

### Transition Systems and Invariants

**Definition 1 (Transition System)** A *transition system*  $P : \langle V, L, l_0, \Theta, \mathcal{T} \rangle$  consists of a set of *variables*  $V$ , a set of *locations*  $L$ , an *initial location*  $l_0$ , an *initial assertion*  $\Theta$  over the variables  $V$ , and a set of *transitions*  $\mathcal{T}$ . Each transition  $\tau \in \mathcal{T}$  is a tuple  $\langle l, l', \rho_\tau \rangle$ , where  $l, l' \in L$  are the *pre* and *post locations*, and  $\rho_\tau$  is the *transition relation*, an assertion over  $V \cup V'$ , where  $V$  represents current-state variables and its primed version  $V'$  represents the next-state variables.

Throughout the paper, unless otherwise stated, we assume that the set of variables  $V = \{x_1, \dots, x_n\}$  is fixed. Furthermore, given an assertion  $\psi$  over the variables  $V$  of a transition system,  $\psi'$  denotes the assertion obtained by replacing each variable  $x \in V$  by  $x' \in V'$ .

The *control-flow graph* (CFG) of a transition system is a graph whose vertices are the locations and whose edges are the transitions. A *path*  $\pi$  of the transition system is a path through its CFG, and the relation  $\rho_\pi$  associated with the path is the composition of the corresponding transition relations.

A *cutset*  $C$  of a transition system  $P$  is a subset of the locations of  $P$  with the property that every cyclic path in  $P$  passes through some location in  $C$ . A location inside a cutset is called a *cutpoint*. A *basic* path  $\pi$  between two cutpoints  $l_i$  and  $l_j$  is a simple path that does not go through any cutpoint, other than the end points.

**Definition 2 (Inductive Assertion Map)** Given a program  $P$  with a cutset  $C$  and an assertion  $\eta_c(l)$ , for each cutpoint  $l$ , we say that  $\eta_c$  is an *inductive assertion map* for  $C$  if it satisfies the following conditions for all cutpoints  $l, l'$ :

**Initiation** For each basic path  $\pi$  from  $l_0$  to  $l$ ,  $\Theta \wedge \rho_\pi \models \eta_c(l)'$ .

**Consecution** For each basic path  $\pi$  from  $l$  to  $l'$ ,  $\eta_c(l) \wedge \rho_\pi \models \eta_c(l')'$ .

Any such partial map may be extended in a natural way to a total map, provided a computable *post image* operator is available. If the cutset  $C$  is obvious from the context, we drop the subscript from the map.

## Linear and Non-linear Constraints

A *linear constraint* over  $V$  is an inequality of the form  $a_1x_1 + \dots + a_nx_n + b \leq 0$ , where  $a_1, \dots, a_n, b$  denote known real-valued coefficients. The constraint is said to be *homogeneous* if  $b = 0$  and *inhomogeneous* otherwise. A *linear assertion* over a set of variables  $V$  is a conjunction of linear constraints over  $V$ . A linear assertion consisting of  $k$  inequalities is said to be *k-linear*. Geometrically, the set of points satisfying a linear assertion forms a *polyhedron*. Linear assertions have been thoroughly studied; problems like satisfiability and projection are known to be decidable [17].

Given a set of vectors  $S$ , the *cone* generated by  $S$ , denoted by  $\text{cone}(S)$ , is the set of all the vectors of the form  $\lambda_1s_1 + \dots + \lambda_ms_m$ , where  $s_1, \dots, s_m \in S$  and  $\lambda_1, \dots, \lambda_m \geq 0$ . A cone is said to be *finitely generated* if it is  $\text{cone}(S)$  for some finite  $S$ . *Polyhedral cones* are cones that can be characterized as the solution spaces of homogeneous linear assertions. A fundamental result states that a cone is finitely generated iff it is polyhedral [17].

A non-linear constraint is an inequality of the form  $P \leq 0$ , where  $P$  is a polynomial on  $x_1, \dots, x_n$ . The degree of the constraint is defined as the degree of the polynomial  $P$ . A *non-linear assertion* is a conjunction of non-linear constraints. A polynomial  $P$  is said to be *reducible* if  $P = P_1P_2$ , where  $P_1$  and  $P_2$  are polynomials of strictly lower degree than  $P$ . The set of points satisfying

<b>integer</b> $i, j$ <b>where</b> $i = 2 \wedge j = 0$	$L = \{l_0, l_1\}, V = \{i, j\},$
$l_0$ : <b>while true do</b>	$\Theta : (i = 2 \wedge j = 0), \mathcal{T} = \{\tau_0, \tau_1, \tau_2\},$
$\left[ \begin{array}{l} i := i + 4 \\ l_1 : \quad \textbf{or} \\ (i, j) := (i + 2, j + 1) \end{array} \right]$	$\tau_0 : \langle l_0, l_1, \text{true} \rangle$ $\tau_1 : \langle l_1, l_0, (i' = i + 4 \wedge j' = j) \rangle$ $\tau_2 : \langle l_1, l_0, (i' = i + 2 \wedge j' = j + 1) \rangle$

**Fig. 1.** A simple program fragment and the corresponding transition system

a non-linear assertion is called a *semi-algebraic* set. Problems such as satisfiability, projection, intersection and union, though still decidable, have a higher complexity than for linear constraints [2].

We say that a transition system is *linear* if its initial assertion  $\Theta$  is linear over  $V$  and its transition relations  $\rho_\tau$  are linear assertions over  $V \cup V'$ . Consequently, for every simple path  $\pi$ , the relation  $\rho_\pi$  is a linear assertion. The rest of this paper deals with linear transition systems. Corresponding to an inductive assertion map, a *k-linear inductive assertion map* ( $k > 0$ ) is defined to be an assertion map, wherein each assertion is *k-linear*.

*Example.* Figure 1 shows a simple program fragment over the variables  $i$  and  $j$ , taken from [7]. As a transition system, it has a two locations  $l_0$  and  $l_1$ , and three transitions  $\tau_0$ ,  $\tau_1$  and  $\tau_2$ .

### Farkas' Lemma

Farkas' lemma provides a sound and complete method for reasoning about systems of linear inequalities.

**Theorem 1 (Farkas' Lemma).** *Consider the following system of linear inequalities over real-valued variables  $x_1, \dots, x_n$ ,*

$$S : \left[ \begin{array}{ccc} a_{11}x_1 + \dots + a_{1n}x_n + b_1 \leq 0 \\ \vdots & \vdots & \vdots \\ a_{m1}x_1 + \dots + a_{mn}x_n + b_m \leq 0 \end{array} \right]$$

*When  $S$  is satisfiable, it entails a given linear inequality*

$$\psi : c_1x_1 + \dots + c_nx_n + d \leq 0$$

*if and only if there exist non-negative real numbers  $\lambda_0, \lambda_1, \dots, \lambda_m$ , such that*

$$c_1 = \sum_{i=1}^m \lambda_i a_{i1}, \quad \dots, \quad c_n = \sum_{i=1}^m \lambda_i a_{in}, \quad d = \left( \sum_{i=1}^m \lambda_i b_i \right) - \lambda_0$$

*Furthermore,  $S$  is unsatisfiable if and only if the inequality  $1 \leq 0$  can be derived as shown above.*

We represent applications of the lemma using a tabular notation:

$$\begin{array}{c|c}
\lambda_0 & -1 \leq 0 \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + b_1 \leq 0 \\
\vdots & \vdots \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + b_m \leq 0
\end{array} \left. \vphantom{\begin{array}{c} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_m \end{array}} \right\} S$$


---


$$\begin{array}{c}
c_1x_1 + \cdots + c_nx_n + d \leq 0 \leftarrow \psi \\
1 \leq 0 \leftarrow \text{false}
\end{array}$$

The antecedents are placed above the line and the consequences below. For each column, the sum of the column entries above the line, with the appropriate multipliers, must be equal to the entry below the line. If a row corresponds to an inequality, the corresponding multiplier is required to be non-negative. This requirement is dropped for rows corresponding to equalities.

### 3 Generating Invariants

We now present our approach to linear invariant generation. We represent the invariant

$$c_1x_1 + \cdots c_nx_n + d \leq 0$$

in terms of its coefficients  $c_1, \dots, c_n, d$ . The main idea behind our technique is to treat these coefficients as unknowns and generate constraints on the coefficients such that any solution corresponds to an inductive assertion. The key to this approach is *Farkas' Lemma*.

Given a transition system and a cutset, we generate a (partial) inductive assertion map  $\eta$  over the cutpoints by encoding initiation and consecution. Let  $\eta(l)$  be represented by the assertion  $c_{l1}x_1 + \cdots + c_{ln}x_n + d_l \leq 0$ , where each  $c_{li}$  and each  $d_l$  is an unknown.<sup>1</sup> The two conditions for the map to be inductive are encoded as follows:

**Initiation:** For each cutpoint  $l$  and each basic path  $\pi$  from  $l_0$  to  $l$ , the path may be an *enabled* path, in which case  $\Theta \wedge \rho_\pi$  is satisfiable, or the path may be *disabled*, in which case,  $\Theta \wedge \rho_\pi$  is unsatisfiable. Initiation can thus be represented by the following table,

$$\begin{array}{c|c}
\lambda_0 & -1 \leq 0 \\
\lambda_1 & a_{11}x_1 + \cdots + a_{1n}x_n + b_1 \leq 0 \\
\vdots & \vdots \\
\lambda_{j-1} & a_{j-1,1}x_1 + \cdots + a_{j-1,n}x_n + b_{j-1} \leq 0 \\
\lambda_j & a_{j1}x_1 + \cdots + a_{jn}x_n + a'_{j1}x'_1 + \cdots + a'_{jn}x'_n + b_j \leq 0 \\
\vdots & \vdots \\
\lambda_m & a_{m1}x_1 + \cdots + a_{mn}x_n + a'_{m1}x'_1 + \cdots + a'_{mn}x'_n + b_m \leq 0
\end{array} \left. \vphantom{\begin{array}{c} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{j-1} \\ \lambda_j \\ \vdots \\ \lambda_m \end{array}} \right\} \begin{array}{l} \Theta \\ \rho_\pi \end{array}$$


---


$$\begin{array}{c}
c_{l1}x'_1 + \cdots + c_{ln}x'_n + d_l \leq 0 \leftarrow \eta(l)' \\
1 \leq 0 \leftarrow \text{disabled}
\end{array}$$

where  $\lambda_0, \dots, \lambda_m \geq 0$ .

<sup>1</sup> We use  $c, d$  with subscripts to denote unknowns and  $a, b$  with subscripts to denote known values.

**Consecution:** For each basic path  $\pi$  from a cutpoint  $l_i$  to a cutpoint  $l_j$ , we encode the consecution condition,  $\eta(l_i) \wedge \rho_\pi \models \eta(l_j)'$ , using Farkas' Lemma. Again there are two cases to consider, one when the path is enabled and the other when it is disabled. The constraints are represented by the table shown below:

$$\begin{array}{c|c}
 \mu & c_{l_i 1}x_1 + \dots + c_{l_i n}x_n + d_{l_i} \leq 0 \leftarrow \eta(l_i) \\
 \lambda_0 & -1 \leq 0 \\
 \lambda_1 & a_{11}x_1 + \dots + a_{1n}x_n + a'_{11}x'_1 + \dots + a'_{1n}x'_n + b_1 \leq 0 \\
 \vdots & \vdots \\
 \lambda_m & a_{m1}x_1 + \dots + a_{mn}x_n + a'_{m1}x'_1 + \dots + a'_{mn}x'_n + b_m \leq 0 \\
 \hline
 & c_{l_j 1}x'_1 + \dots + c_{l_j n}x'_n + d_{l_j} \leq 0, \leftarrow \eta(l_j)' \\
 & 1 \leq 0 \leftarrow \text{disabled}
 \end{array} \left. \vphantom{\begin{array}{c} \mu \\ \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_m \end{array}} \right\} \rho_\pi$$

where  $\mu, \lambda_0, \dots, \lambda_m \geq 0$ .

The constraints corresponding to initiation are linear, as are the constraints corresponding to the disabled case of consecution. However, the constraints for the enabled case of consecution are non-linear due to the presence of the multiplier  $\mu$  in a row containing unknown coefficients. Methods for solving these constraints are discussed in Section 4.

*Example.* Consider the program shown in Figure 1. All cycles of the program are cut by  $l_0$ . Let  $\varphi : c_1i + c_2j + d \leq 0$  be the target invariant at this cutpoint. The initial condition,  $i = 2 \wedge j = 0$ , generates the constraints

$$\begin{array}{c|c}
 \lambda_0 & -1 \leq 0 \\
 \lambda_1 & i - 2 = 0 \\
 \lambda_2 & j = 0 \\
 \hline
 & c_1i + c_2j + d \leq 0 \leftarrow \varphi \\
 & 1 \leq 0 \leftarrow \text{disabled}
 \end{array}$$

We can ignore the disabled case since the initial condition is satisfiable. With the requirement that  $\lambda_0$  be non-negative, we obtain the following constraints:

$$\psi_\Theta : (\exists \lambda_0, \lambda_1, \lambda_2) \left[ \begin{array}{l} c_1 = \lambda_1 \wedge c_2 = \lambda_2 \wedge \\ d = -2\lambda_1 - \lambda_0 \wedge \lambda_0 \geq 0 \end{array} \right] \quad (1)$$

Two paths cycle back to  $l_0$ : path  $\pi_1$  using  $\tau_1$  and  $\pi_2$  through  $\tau_2$ . For the path  $\pi_1$ , the transition relation  $\rho_{\pi_1}$  is given by  $i - i' + 4 = 0 \wedge j - j' = 0$ . The two constraints for consecution are given by the table

$$\begin{array}{c|c}
 \mu & c_1i + c_2j + d \leq 0 \\
 \lambda_0 & -1 \leq 0 \\
 \lambda_1 & i - i' + 4 = 0 \\
 \lambda_2 & j - j' = 0 \\
 \hline
 & c_1i' + c_2j' + d \leq 0 \leftarrow \varphi \\
 & 1 \leq 0 \leftarrow \text{disabled}
 \end{array}$$

resulting in the constraints:

$$(\exists \bar{\lambda}, \mu) \left[ \begin{array}{l} 0 = \mu c_1 + \lambda_1 \wedge \\ 0 = \mu c_2 + \lambda_2 \wedge \\ c_1 = -\lambda_1 \wedge c_2 = -\lambda_2 \wedge \\ d = \mu d + 4\lambda_1 - \lambda_0 \wedge \\ \mu, \lambda_0 \geq 0 \end{array} \right] \vee (\exists \bar{\lambda}, \mu) \left[ \begin{array}{l} 0 = \mu c_1 + \lambda_1 \wedge \\ 0 = \mu c_2 + \lambda_2 \wedge \\ 0 = -\lambda_1 \wedge 0 = -\lambda_2 \wedge \\ 1 = \mu d + 4\lambda_1 - \lambda_0 \wedge \\ \mu, \lambda_0 \geq 0 \end{array} \right] \quad (2)$$

Similarly, the constraints corresponding to consecution for  $\pi_2$  are

$$(\exists \bar{\lambda}, \mu) \left[ \begin{array}{l} 0 = \mu c_1 + \lambda_1 \wedge \\ 0 = \mu c_2 + \lambda_2 \wedge \\ c_1 = -\lambda_1 \wedge c_2 = -\lambda_2 \wedge \\ d = \mu d + 2\lambda_1 + \lambda_2 - \lambda_0 \wedge \\ \mu, \lambda_0 \geq 0 \end{array} \right] \vee (\exists \bar{\lambda}, \mu) \left[ \begin{array}{l} 0 = \mu c_1 + \lambda_1 \wedge \\ 0 = \mu c_2 + \lambda_2 \wedge \\ 0 = -\lambda_1 \wedge 0 = -\lambda_2 \wedge \\ 1 = \mu d + 2\lambda_1 + \lambda_2 - \lambda_0 \wedge \\ \mu, \lambda_0 \geq 0 \end{array} \right] \quad (3)$$

We will eliminate the quantifiers and complete the example in Section 4. The technique presented in this section can be easily extended to  $k$ -linear assertion maps. Farkas' Lemma may be used in this case to obtain constraints for initiation and consecution with few changes.

**Theorem 2.** *For any  $k > 0$ , a  $k$ -linear assertion map  $\eta$  is inductive iff it is a solution to the system of constraints generated by our method.*

The theorem follows directly from Farkas' Lemma and our constraint generation technique. It states that our technique is sound. Furthermore, completeness holds for those linear invariants that can be proved using linear inductive assertions. A linear program may have a reachable state-space that is not convex, but inductive linear assertions can only characterize convex sets. Hence, there are linear programs which satisfy linear invariants that can only be established by resorting to non-linear or non-convex inductive assertions. This result is demonstrated by the work of Clarke [3].

In theory, having a conjunction of  $k > 1$  linear inequalities at each cutpoint is better than a single inequality at each cutpoint. In practice however, we find that the constraints obtained for  $k > 1$  are too complex to solve exactly for all but the smallest of systems. Nevertheless, as Section 5 illustrates, our technique with exact solution is powerful even when restricted to 1-linear assertions. Furthermore,  $k$ -linear inductive assertions can be approximated iteratively, by starting with 1-linear assertions and strengthening each transition relation with the invariants computed in the previous stage.

## 4 Solving Constraints

Our method extracts linear and non-linear constraints characterizing inductive assertions. Any solution to these constraints is thus inductive. These constraints may be solved by eliminating the quantified variables. In practice, however, quantifier elimination is a costly process with exponential time complexity. Therefore, we exploit various techniques like factorization and root finding to simplify the constraints, thereby reducing the size of each quantifier elimination instance and

that of the result after elimination. In some cases, we are able to generate all the solutions solely by making use of these simplification techniques, without resorting to quantifier elimination. We present some of these techniques and then apply them to solve the constraints generated for our running example. The reader is referred to the comprehensive survey by Bockmayr and Weispfenning [2] on constraint solving for more details.

## Linear Constraints

Geometrically, the set of points satisfying an inhomogeneous linear assertion forms a *polyhedron*, and the set satisfying a homogeneous linear assertion is a *polyhedral cone*; elimination of variables corresponds to *projection*. One approach to projection is to compute the generators of the polyhedron and then project these generators on to the free variables. These generators can be computed by the *simplex* method [17] or the *double description* method [9]. In our examples, presented in the next section, we use an implementation of the double description method called POLKA [10].

Alternatively, projection and the computation of generators can be achieved through a quantifier elimination method called *Fourier's* elimination, which eliminates variables from the system of constraints incrementally [2]. Due to its simplicity, Fourier's elimination has been used widely to solve linear constraints, even though its complexity is exponential.

## Non-linear Constraints

Non-linear constraints can be solved by direct quantifier elimination or indirect methods using techniques such as factorization and polynomial root solving.

The original breakthrough in quantifier elimination was due to Tarski [18]. However, it was not computationally feasible until Collins introduced *Cylindrical Algebraic Decomposition* [5]. Recently, there have been many practical implementations based on this technique. Notable among them is QEPCAD, which incorporates many improvements to the original CAD algorithm [4]. The time complexity of the algorithm is exponential in the size of the formula. However, the running time can be brought down significantly for low degree polynomials using the elimination at test points method due to Weispfenning [19]. After quantifier elimination, simplification is carried out using factorization and Gröbner Bases. This method has been implemented in REDLOG [8].

Another approach is to use linear programming and delay processing non-linear constraints until they can be linearized or simplified to an extent that they can be solved directly. This approach, which is at the heart of many CLP based solvers [15], works for a surprisingly large variety of problems. However, there are problems that require non-linear constraint solving. Thus, non-linear constraint solvers have been incorporated into the CLP paradigm. For instance, the RISC-CLP(R) system uses quantifier elimination in the background to solve constraints [14].

## Heuristics

The constraints we obtain are of a low degree and hence, REDLOG is the most suitable tool. A disadvantage of using quantifier elimination is the size of the result after elimination. In general, the final result after elimination is a boolean combination of, mostly unfactorized, polynomial equalities and inequalities, containing redundant non-linear inequalities that need to be detected and removed. It has been our experience so far that the majority of these reduce to linear factors, and that non-linear irreducible polynomials are rare.

If the polynomials in the result are linear, then the cone containing all the solutions is polyhedral, and a minimal set of generators for this cone can be computed using the double description method. The set of inductive invariants can be completely characterized by considering each of these generators as a constraint.

If there are irredundant and unfactorizable non-linear constraints, the cone of consequences may or may not be finitely generated. However, we are unaware of an efficient method for deciding which case holds. When the cone is not finitely generated, it is not possible to characterize all invariants without using a parametric representation.

We have found a few heuristics that are effective in dealing with non-linear factors: If a polynomial  $P$  is reducible, it is always possible to reduce the degree of the constraints by splitting. For instance, the constraint  $P_1 P_2 \leq 0$  is equivalent to the disjunction  $(P_1 \leq 0 \wedge P_2 \geq 0) \vee (P_1 \geq 0 \wedge P_2 \leq 0)$ . Further information about polynomial factorization can be found in standard textbooks on the topic [21].

Another heuristic, especially effective when the result of the elimination is too large to be factorized or simplified, is to set some of the coefficients to zero, in effect restricting the target invariants to those involving only a subset of the variables of the program. Furthermore, since any two-dimensional cone is finitely generated, setting sufficiently many variables to zero always yields a polyhedral cone.

*Example.* We can now complete our example from the previous section by applying the techniques mentioned in this section to solve the constraints.

The initiation constraint shown in (1) is linear and simplifies to  $2c_1 + d \leq 0$ . The constraints corresponding to consecution for path  $\pi_1$  are shown in (2). It is possible to solve them using quantifier elimination. However, a simple substitution on the first clause yields  $-\mu\lambda_1 + \lambda_1 = 0 \wedge -\mu\lambda_2 + \lambda_2 = 0$ , which can be simplified to  $\mu = 1$  or  $\lambda_1 = \lambda_2 = 0$ . Branching on both possibilities, we obtain  $c_1 \leq 0$  for the former and  $c_1 = c_2 = 0, d \leq 0$  for the latter. The other clause can be similarly solved using substitution, yielding  $c_1 = c_2 = 0 \wedge d \geq 0$ . Consecution for path  $\pi_2$  can also be solved using factorization. The final result is:

$$(2c_1 + d \leq 0) \wedge \left[ \begin{array}{l} (c_1 = c_2 = 0 \wedge d \leq 0) \vee \\ (c_1 \leq 0) \vee \\ (c_1 = c_2 = 0 \wedge d \geq 0) \end{array} \right] \wedge \left[ \begin{array}{l} (c_1 = c_2 = 0 \wedge d \leq 0) \vee \\ (2c_1 + c_2 \leq 0) \vee \\ (c_1 = c_2 = 0 \wedge d \geq 0) \end{array} \right]$$



```

integer  $i, j, k$  where  $i = 1 \wedge j = 1 \wedge 0 \leq k \leq 1$ 
 $l_0$  : while true do
     $l_1$  :  $(i, j, k) := (i + 1, j + k, k - 1)$ 
    
```

**Fig. 2.** INCREMENT

After converting this into DNF and eliminating unsatisfiable disjuncts, we obtain the following generators shown along with the corresponding invariants:

$c_1$	$c_2$	$d$	$c_1 i + c_2 j + d \leq 0$
0	0	-1	$-1 \leq 0$
0	-1	0	$-j \leq 0$
-1	2	2	$-i + 2j + 2 \leq 0$

These match the invariants obtained in [7]. Additional inductive assertions, such as  $-i + 2 \leq 0$  can be obtained as consequences of the assertions shown above.

## 5 Applications

We now demonstrate our approach on several examples.

### Increment

Consider the program INCREMENT shown in Figure 2. With each iteration it increments  $i$ , while manipulating the variables  $j$  and  $k$ . Surprisingly, the analysis of Cousot and Halbwachs [7] misses the obvious invariant  $i \geq 1$ . On the other hand, our method produces the invariants  $1 \leq i + k \leq 2$  and  $i \geq 1$  in one iteration. This phenomenon, in which the presence of additional, independent variables weakens the invariants generated using abstract interpretation, can be observed in more realistic programs, e.g. the implementation of MERGESORT presented by Wirth [20].

### Heapsort

We applied our method to HEAPSORT, shown in Figure 3 and taken from [7]. Arrays and operations involving arrays were not modeled in the transition system, and branches involving array conditions were treated as non-deterministic choices. All cycles are cut by  $l_3$ . There are eight paths that go from  $l_3$  back to itself. Upon simplification, the formula obtained after elimination does not contain any non-linear constraint, and the following invariants were easily extracted:

$$\begin{array}{lll}
 l - 1 \geq 0 & r \geq 2 & 2l - r \geq 0 \\
 n \geq 3 & r \leq n & j = 2i
 \end{array}$$

Repeating the analysis assuming  $l \geq 1$ , we obtained the additional invariants  $2l \leq j$ ,  $2l + 2r \geq j + 2$ , matching the invariants generated using abstract interpretation [7].

```

integer  $n, l, r, i, j$  where  $l = \frac{n}{2} + 1 \wedge n \geq 2 \wedge r = n$ 
realarray  $T[1 \dots n]$ ;
real  $k$ ;
 $l_0$  : if  $l \geq 2$  then
     $l_0^a$  :  $(l, k) := (l - 1, T[l]);$ 
else
     $l_0^b$  :  $(k, T[r], r) := (T[r], T[1], r - 1);$ 
end if
 $l_1$  : while  $r \geq 2$  do
     $l_2$  :  $(i, j) := (l, 2l);$ 
     $l_3$  : while  $j \leq r$  do
         $l_4$  : if  $j \leq r - 1 \wedge T[j] < T[j + 1]$  then
             $l_4^a$  :  $j := j + 1;$ 
        end if
         $l_5$  : if  $k \geq T[j]$  then
             $l_5^a$  : break;
        end if;
         $l_6$  :  $(T[i], i, j) := (T[j], j, 2j);$ 
    end while
     $l_7$  :  $T[i] := k;$ 
     $l_8$  : if  $l \geq 2$  then
         $l_8^a$  :  $(l, k) := (l - 1, T[l]);$ 
    else
         $l_8^b$  :  $(k, T[r], r) := (T[r], T[1], r - 1);$ 
    end if
     $l_9$  :  $T[1] := k;$ 
end while

```

Fig. 3. HEAPSORT

## Vagrant Robot

We analyzed a hybrid system modeling the position of a robot over time, taken from [12], represented as the linear transition system presented in Figure 4. The robot works in two alternating phases modeled by locations  $l_0$  and  $l_1$ , each taking between 1 and 2 seconds. In  $l_0$  it moves approximately northeast (both  $x$  and  $y$  increase), and in  $l_1$  it moves approximately southeast ( $x$  increases and  $y$  decreases). The result after quantifier elimination is too large to be factorized or to be converted into some normal form by REDLOG. Therefore, we analyzed the expression with each of the coefficients set to 0, thus looking for invariants involving at most two variables at a time. The invariants obtained for cutpoint  $l_0$  were:  $t \leq x \leq 2t$ , and  $-t \leq y \leq t$ . Taking their post image produces the following invariants at location  $l_1$ :  $t \leq x \leq 2t$ ,  $-t + 2 \leq y \leq t + 2$ ,  $y \leq 2t$  and  $t \geq 1$ .

Note that the invariant assertions above are true immediately after a discrete mode transition is taken. If the continuous evolution is to be taken into account, as is always the case for a hybrid system, we need to perform an additional post

$$\begin{aligned}
 V &= \{x, y, t\}, L = \{l_0, l_1\}, T = \{\tau_1, \tau_2\} \\
 \tau_1 &: \langle l_0, l_1, \rho_1 \rangle, \tau_2 : \langle l_1, l_0, \rho_2 \rangle \\
 \Theta &: x = 0 \wedge y = 0 \wedge t = 0 \\
 \rho_1 &: \left[ \begin{array}{l} t' - t \leq x' - x \leq 2(t' - t) \wedge \\ t' - t \leq y' - y \leq 2(t' - t) \wedge \\ 1 \leq t' - t \leq 2 \end{array} \right] \quad \rho_2 : \left[ \begin{array}{l} t' - t \leq x' - x \leq 2(t' - t) \wedge \\ -(t' - t) \geq y' - y \geq -2(t' - t) \wedge \\ 1 \leq t' - t \leq 2 \end{array} \right]
 \end{aligned}$$

**Fig. 4.** Transition system for the vagrant robot

image computation at each location to obtain the true invariant from a hybrid system point of view. In either case, the invariants above suffice to establish the unreachability of the point  $y = 12$ ,  $x = 9$ , which is posed as a challenge to the reader in [12].

## 6 Conclusions

We have presented a method that generates invariants by solving for their coefficients directly. This is in contrast to traditional methods, which work iteratively toward a fixed point. The method generates constraints using *Farkas' Lemma*. Theoretically, the method is sound and complete, guaranteeing that any linear invariant of a linear program which is provable using an inductive linear assertion can be found using our approach.

The main drawback of the method is that it produces non-linear constraints. As a result, while the technique can be applied to generate subtle invariants for small systems, its applicability to larger systems is limited. We are confident that, as the state of art in non-linear constraint solving advances, our method will become more and more practical.

## References

1. F. Besson, T. Jensen, and J.-P. Talpin. Polyhedral analysis of synchronous languages. In *Static Analysis Symposium, SAS'99*, Lecture Notes in Computer Science 1694, pages 51–69, 1999.
2. A. Bockmayr and V. Weispfenning. Solving numerical constraints. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 12, pages 751–842. Elsevier Science, 2001.
3. E.M. Clarke. Synthesis of resource invariants for concurrent programs. In *ACM Principles of Programming Languages*, pages 211–221, January 1979.
4. G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, sep 1991.
5. G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In H.Brakhage, editor, *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183, 1975.

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 Principles of Programming Languages*, pages 238–252, 1977.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among the variables of a program. In *ACM Principles of Programming Languages*, pages 84–97, January 1978.
8. A. Dolzmann and T. Sturm. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.
9. K. Fukuda and A. Prodon. Double description method revisited. In *Combinatorics and Computer Science*, volume 1120 of *LNCS*, pages 91–111. Springer-Verlag, 1996.
10. N. Halbwachs and Y.-E. Proy. *POLyhedra desK cAlculator (POLKA)*. VERIMAG, Montbonnot, France, September 1995.
11. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
12. T.A. Henzinger and P.-H. Ho. Model-checking strategies for hybrid systems. In *Conference on Industrial and Engineering Applications of AI and Expert Systems*, 1994.
13. T.A. Henzinger and P.-H. Ho. Algorithmic analysis of nonlinear hybrid systems. In P. Wolper, editor, *Computer-Aided Verification*, LNCS 939, pages 225–238. 1995.
14. H. Hong. RISC-CLP(Real): Constraint logic programming over real numbers. In *CLP: Selected Research*. MIT Press, 1993.
15. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Principles of Programming Languages(POPL)*, pages 111–119, January 1987.
16. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
17. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1986.
18. A. Tarski. A decision method for elementary algebra and geometry. *Univ. of California Press, Berkeley*, 5, 1951.
19. V. Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. In *Applied Algebra and Error-Correcting Codes (AAECC) 8*, pages 85–101, 1997.
20. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.
21. R. Zippel. *Effective Polynomial Computation*. Kluwer Academic Publishers, Boston, 1993.

# To Store or Not to Store

Gerd Behrmann<sup>1</sup>, Kim G. Larsen<sup>1</sup>, and Radek Pelánek<sup>\*2</sup>

<sup>1</sup> Aalborg University, Frederik Bajers Vej 7E, 9220 Aalborg, Denmark

<sup>2</sup> Masaryk University Brno, Czech Republic

**Abstract.** To limit the explosion problem encountered during reachability analysis we suggest a variety of techniques for reducing the number of states to be stored during exploration, while maintaining the guarantee of termination and keeping the number of revisits small. The techniques include static analysis methods for component automata in order to determine small sets of covering transitions. We carry out extensive experimental investigation of the techniques within the real-time verification tool UPPAAL. Our experimental results are extremely encouraging: a best combination is identified which for a variety of industrial case-studies reduces the space-consumption to less than 10% with only a moderate overhead in time-performance.

**Keywords.** Timed automata model checking, Static analysis

## 1 Introduction

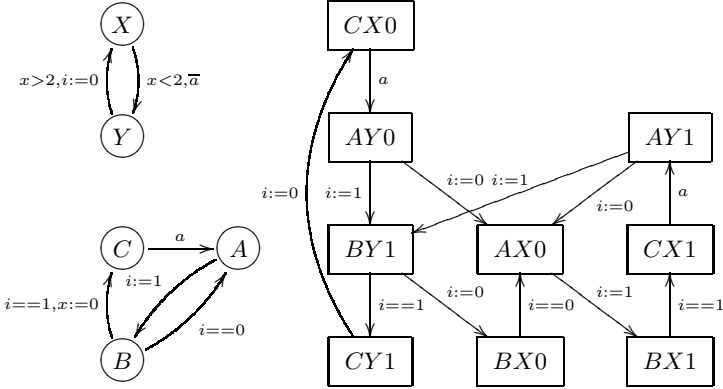
Reachability analysis has proved one of the most successful methods for automated analysis of concurrent systems. Several verification problems, e.g. trace-inclusion, invariant checking and model checking of temporal logic formula using test automata may be solved using reachability analysis. However the major problem in applying reachability analysis is the potential combinatorial explosion in the size of state spaces and the resulting excessive memory requirements. During the last decade numerous symbolic and reduction techniques have been put forward [5,7,13,3,12,10] in order to avoid this explosion problem, playing a crucial role in the successful development of a number of verification tools for finite-state and timed systems (*e.g.* SMV, SPIN, visualSTATE, UPPAAL, KRONOS).

The explosion in memory consumption is closely linked to the need for storing states during exploration, primarily with the aim of guaranteeing termination but also to avoid repeated exploration (revisits) of states. However, one may maintain the guarantee of termination while keeping the number revisits small without necessarily storing *all* states. As an example consider the state space of the network in Fig. 1. Here, termination is guaranteed with storing of only two states (*e.g.* AY0 and AY1). The main question addressed in this paper is how to efficiently decide whether “to store” or “not to store” states during exploration.

In answering this question, we see that it suffices to store enough states so that all cycles in the global state space are covered. However, as finding

---

\* Supported by GA ČR grant no. 201/03/0509



**Fig. 1.** A network of two timed automata and its discrete state space.

the smallest such set is NP-complete, we instead propose a number of efficient storing strategies yielding small covering sets. The techniques include storage only of every  $k$ -th state along a path, and storage only of states with multiple successors. Also, static analysis methods of cycles of the individual automata in a network are given in order to determine a (small) set of covering transitions based partially on a heuristic analysis of the quality of such sets using a (cheap) random walk analysis.

Within the real-time verification tool UPPAAL, we conduct a thorough experimental investigation of the various storing techniques identifying their best combinations. Our experimental results are extremely encouraging: for the majority of a range of industrial case-studies and examples studied in the literature it is possible to reduce the number of states stored to less than 10% compared with that of the currently distributed version of UPPAAL and with only a small overhead in time-performance.

The outline of the paper is as follows: section 2 introduces the timed automata network model used in UPPAAL together with the basic reachability algorithm. Section 3 provides a number of revised reachability algorithms based on different storing strategies. Section 4 revises the notion of covering set and provides a number of static analysis algorithms for identifying small such sets. Section 5 provides an extensive experimental analysis of the reduction techniques suggested, and, finally, section 6 contains the conclusion.

## 2 Preliminaries

Let  $X$  be a set of clocks,  $V$  a set of bounded integer variables and  $\Sigma$  a set of actions and co-actions s.t.  $a = \bar{a}$  and  $a \in \Sigma \Leftrightarrow \bar{a} \in \Sigma$ . An integer expression over  $V$  is an expression on the form  $v$ ,  $c$ ,  $e_1 + e_2$  or  $e_1 - e_2$ , where  $v \in V$ ,  $c$  is an integer constant, and  $e_1$  and  $e_2$  are integer expressions. Let  $G(X, V)$  be the set

of all guards, s.t.  $x \bowtie c$ ,  $e_1 \bowtie e_2$ , and  $g_1 \wedge g_2$  are guards, where  $x$  is a clock,  $c$  is an integer over  $V$ ,  $e_1$  and  $e_2$  are integer expressions,  $g_1$  and  $g_2$  are guards, and  $\bowtie \in \{<, \leq, =, \geq, >\}$ . Let  $U(X, V)$  be the set of updates, s.t.  $x := 0$ ,  $v := e$ , and  $u_1$ ;  $u_2$  are updates, where  $x$  is a clock,  $v$  is a bounded integer variable,  $e$  is an integer expression, and  $u_1$  and  $u_2$  are updates.

**Definition 1 (Timed Automata).** A *timed automaton*  $\mathcal{A}$  over  $X$ ,  $V$  and  $\Sigma$  is a tuple  $(L, l^0, E, \text{guard}, \text{sync}, \text{up}, I)$  where  $L$  is a finite set of locations;  $l^0 \in L$  is the initial location;  $E \subseteq L \times L$  is the set of edges;  $\text{guard} : E \rightarrow G(X, V)$ ,  $\text{sync} : E \rightarrow \Sigma \cup \{\tau\}$ ,  $\text{up} : E \rightarrow U(X, V)$  assign guards, actions, and updates to edges; and  $I : L \rightarrow G(X, V)$  assigns invariants to locations. A *network of timed automata* is a tuple of timed automata  $A_1 \parallel \dots \parallel A_n$  over  $X$ ,  $V$ , and  $\Sigma$ .

Notice that the terms *location* and *edge* refer to the syntactic elements of a timed automaton. A *concrete state* of a timed automaton is a semantic element and is defined as a triple  $(l, u, \gamma)$  of the current location  $l$ , and two functions  $u$  and  $\gamma$  assigning values to clocks and variables, respectively. More formally, let  $\Gamma$  be a function from  $V$  to intervals in  $\mathbb{Z}$  s.t.  $\forall v \in V : 0 \in \Gamma(v)$ , thus defining the range of the variables. A clock valuation,  $u$ , is a function from  $X$  to the non-negative reals,  $\mathbb{R}_{\geq 0}$ . A variable valuation,  $\gamma$ , is a function from  $V$  to  $\mathbb{Z}$  such that  $\gamma(v) \in \Gamma(v)$ . Let  $\mathbb{R}^X$  be the set of all clock valuations and  $\mathbb{Z}^V$  the set of all integer valuations. We write  $(u, \gamma) \models g$  when the guard  $g$  is satisfied in the context of  $u$  and  $\gamma$ .

The semantics of (a Network of) Timed Automata is often stated in terms of a labelled transition system  $\rightarrow$  with  $(L_1 \times \dots \times L_n) \times \mathbb{R}^X \times \mathbb{Z}^V$  being the set of states. There are two types of transitions: Whenever allowed by the invariant of the current locations, time can pass. The resulting *delay transitions* do not change the locations nor the variables, but all clocks are incremented by the exact same delay. Alternatively, if the guard of an edge (or two edges if they synchronise via complementing actions) is satisfied, we might trigger an *edge transition*. Edge transitions do not take time, i.e. the clock valuation is not changed except to reflect updates directly specified on the edge involved. The same is true for the variable valuation.

Given a state property  $\varphi$ , the reachability problem for timed automata is that of deciding whether there is a path from the initial state to a state satisfying  $\varphi$ . A standard forward breadth-first or depth-first search based directly on the semantics of a timed automaton is unlikely to terminate, since the state-space is uncountably infinite. It is well known that several exact and finite abstractions based on *zones* exist, see [1,2]. Zones are sets of clock valuations definable by conjunctions of constraints of the forms  $x \bowtie c$  and  $x - y \bowtie c$ , where  $x$  and  $y$  are clocks and  $c$  is an integer. Using zones, one can define the *symbolic semantics* of a network of timed automata as a labelled transition relation  $\Rightarrow$  over *symbolic states* on the form  $(l, Z, \gamma)$ , where  $l$  is a location,  $Z$  is a zone, and  $\gamma$  is a variable valuation, s.t.  $(l, Z, \gamma) \xRightarrow{t} (l', Z', \gamma')$  iff  $\forall \sigma' \in Z' : \exists \sigma \in Z : (l, \sigma, \gamma) \xrightarrow{t, \delta} (l', \sigma', \gamma')$  ( $t$  being a set of edges and  $\delta$  being a non-negative real-valued delay). We will skip the formal definition and assume the existence of such a symbolic semantics. In the following, we will refer to symbolic states simply as states.

```

proc REACHABILITY
   $W = \{init\_state\}; P = \emptyset;$ 
  while  $W \neq \emptyset$  do
    get  $s$  from  $W$ ;
    if  $s \models \varphi$  then return true; fi
     $P = P \cup \{s\};$ 
    foreach  $s', t : s \xrightarrow{t} s'$  do
      if  $s' \notin P \cup W$  then  $W = W \cup \{s'\};$  fi od
    od
  return false;
end

```

**Fig. 2.** Decision procedure for the reachability problem.

Given the symbolic semantics, defining a decision procedure for the reachability problem is easy and is shown in Fig. 2. Here,  $P$  is the set of already visited states (the passed list), and  $W$  is the set of states which are to be explored (the waiting list). The passed list is usually implemented as a hash table, and the waiting list is often a queue or a stack.

### 3 Storing Strategies

By storing states in a passed list, we avoid that states are explored more than once, thus ensuring termination and efficiency. For termination, not all states need to be stored. Theoretically, only enough states such that all cycles in the symbolic transition system are covered must be stored – this is called a *feedback vertex set*. However, finding the smallest such set is NP-complete [8]. Our goal in this paper is to find efficient strategies to obtain small feedback vertex sets for the symbolic transition relation (thus ensuring termination), while keeping the number of revisited states as low as possible.

We start by presenting a revised version of the reachability algorithm, see Fig. 3, which adds an extra integer field, *flag*, to each state in the waiting list. The interpretation of this flag depends on the two functions  $to\_store(s, flag)$  and  $next\_flag(s, t, flag)$ , which compute whether to store  $s$  and what the flag of a successor of  $s$  should be, respectively. The reductions presented in the following are all instances of this generic scheme and differ only in the actual definition of  $to\_store$  and  $next\_flag$ .

**Distance** In many real life examples, cycles are rather long and it is sufficient to store every  $k$ -th state along any path, *i.e.*

$$\begin{aligned}
 to\_store(s, flag) &\equiv flag \bmod k = 0 \\
 next\_flag(s, t, flag) &\equiv flag + 1
 \end{aligned}$$

For the example in Fig. 1 this strategy stores (provided that we use BF-Search which starts at CX0 with flag 0) for  $k = 2$ : {CX0, BY1, AX0, CX1}, and for  $k = 3$ : {CX0, CY1, BX0, BX1}.



**proc** REACHABILITY-WITH-REDUCTION

$W = \{(init\_state, 0)\}; P = \emptyset;$

**while**  $W \neq \emptyset$  **do**

  get  $(s, flag)$  from  $W$ ;

**if**  $s \models \varphi$  **then** return true; **fi**

**if**  $to\_store(s, flag)$  **then**  $P = P \cup \{s\}$ ; **fi**

**foreach**  $s', t : s \xrightarrow{t} s'$  **do**

**if**  $s' \notin P \cup W$  **then**  $W = W \cup \{(s', next\_flag(s, t, flag))\}$ ; **fi od**

**od**

  return false;

**end**

**Fig. 3.** Revised decision procedure for the reachability problem, with hooks for adding various reduction techniques.

**Successors** Observations of state spaces of industrial protocols revealed that there are usually chains of states with only one successor. It is practically useless to store several states from such a chain. Thus only states which have more than one successor need to be stored. To avoid that cycles of states with only one successor cause problems, we use the flag as a counter to store at least one state of such cycles, *i.e.*

$$to\_store(s, flag) \equiv |succ(s)| > 1 \vee flag = k$$

$$next\_flag(s, t, flag) \equiv \begin{cases} 0 & \text{if } to\_store(s, flag) \\ flag + 1 & \text{otherwise} \end{cases}$$

In practice these cycles are infrequent and it is safe to use a large value for  $k$  (in none of our numerous experiments we considered such cycles occurred). For the example in Fig. 1 this strategy stores: {AY0, BY1, AY1}.

**Covering Set** In [11] a strategy based on static analysis of the cycles of the individual automata in the system was proposed. Here we provide a generalisation of the idea, inspired by [9]. Suppose that we have a set of edges, *Cover*, with the property that each cycle in the global state space contains at least one edge from *Cover*. Then it is sufficient to store states that are targets of transitions derived from edges in *Cover*, *i.e.*

$$to\_store(s, flag) \equiv flag = 1$$

$$next\_flag(s, t, flag) \equiv \begin{cases} 1 & \text{if } t \in Cover \\ 0 & \text{otherwise} \end{cases}$$

We will consider the problem of finding *Cover* in the next section. Referring to the example of Fig. 1, there are three cycles: (CX0, AY0, BY1, CY1), (BY1, BX0, AX0, BX1, CX1, AY1), and (AX0, BX1, CX1, AY1). Possible candidates for *Cover* are { (Y,X) } and { (C,A) }, resulting in the states in {AX0, CX0, BX0} and {AY0, AY1} being stored, respectively.

**Random** The simplest way to decide which states to store is to use randomness. In this case the flag is actually not needed since we simply store states with some probability,  $p$ . A deterministic version of this strategy is to use a global counter, by counting the number of newly visited states and store each  $k$ -th state ( $k \approx \frac{1}{p}$ ).

**Combinations** There are many possibilities for combining the previous strategies. One particular combination of *covering set*, *successors* and *distance* reductions is the following:

$$\begin{aligned} to\_store(s, flag) &\equiv flag \geq k \wedge |succ(s)| > 1 \vee flag = k^2 \\ next\_flag(s, t, flag) &\equiv \begin{cases} 0 & \text{if } to\_store(s, flag) \\ flag + 1 & \text{if } \neg to\_store(s, flag) \wedge t \in Cover \\ flag & \text{otherwise} \end{cases} \end{aligned}$$

We increase *flag* when the state is a target of covering edge. And we store states only if the flag is greater than  $k$  and the number of successors is greater than one or the flag is  $k^2$  (the latter catches cycles where each state only has one successor). There are of course many other possible combinations. We have tried other combinations of covering set, successors, distance, and random strategies with non-uniform probabilities. However, the behaviour of these combinations was usually similar to the one presented above.

**Lemma 1.** *Procedure REACHABILITY-WITH-REDUCTION terminates with any of these storing strategies (resp. with probability one for random strategies).*

## 4 Covering Set

In this section we discuss the problem of finding a *covering set*, being a set of edges with the property that each cycle in the symbolic state space uses at least one edge from this set. The algorithms used are due to [9], where they were used in the context of partial order reduction. We contribute additional methods for identifying relations among cycles which lead to smaller covering sets and a heuristic analysis of the quality of a covering set based on random walk analysis.

Referring to the example in Fig. 1, possible candidates for *Cover* are  $\{(Y, X)\}$  and  $\{(C, A)\}$ . The challenge we face is to compute such a covering set for a network based solely on a static analysis of the control structure of each automaton, *i.e.*, without unfolding the symbolic transition relation. Elementary cycles in the control structure, henceforth called *local cycles*, are important elements of such an analysis, since cycles in the symbolic state space, henceforth called *global cycles*, are formed by unfolding local cycles. The straightforward way of finding a covering set is to compute all local cycles and pick an edge from each. In the example, there are three local cycles:  $\{X, Y\}$ ,  $\{A, B, C\}$ , and  $\{A, B\}$ , leading to a covering set of  $\{(Y, X), (A, B)\}$ , for instance. But if we look closer, it becomes clear that the cycle  $\{X, Y\}$  cannot be realised without the synchronisation provided by  $\{A, B, C\}$ , and that  $\{A, B\}$  cannot be realised without the

reset provided by  $\{X, Y\}$ . We say that  $\{A, B\}$  is *covered by*  $\{X, Y\}$ . We will now formalise this.

For the rest of this section we assume the existence of a network of  $n$  timed automata,  $A_i = (L_i, l_0, E_i, \text{guard}_i, \text{sync}_i, \text{up}_i, I_i)$ . A *local cycle* of  $A_i$  is a sequence  $(e_1, \dots, e_m)$  of edges in  $E_i$  iff there is a set of locations  $l_1, \dots, l_m$  in  $L_i$  s.t.  $l_1 \xrightarrow{e_1}_i \dots \xrightarrow{e_{m-1}}_i l_m \xrightarrow{e_m}_i l_1$ , where  $l \xrightarrow{e}_i l' \Leftrightarrow e = (l, l') \wedge e \in E_i$ . A *global cycle* is a sequence  $(t_1, \dots, t_m)$ , where each  $t_i$  is a set of edges, iff there is a sequence of symbolic states,  $s_1, \dots, s_m$  s.t.  $s_1 \xrightarrow{t_1} \dots s_m \xrightarrow{t_m} s_1$ , where  $\Rightarrow$  is the labelled transition relation of the network. A *projection*  $P(o)$  of a global cycle  $o = (t_1, \dots, t_m)$  is the set of local cycles corresponding to subsequences of  $o$ , i.e.,  $c \in P(o)$  iff  $c = (e_1, \dots, e_k)$  is a local cycle and  $\exists 1 \leq i_1 < \dots < i_k \leq m, \forall 1 \leq j \leq k : e_j \in t_{i_j}$ . By an abuse of notation we sometimes treat sequences as sets.

A set of transitions  $T$  *covers* a set of cycles  $C$  iff for each cycle  $c \in C : T \cap c \neq \emptyset$ . A set of transitions  $T$  is a *covering set* of a network  $N$  iff  $T$  covers all global cycles of  $N$ . The straightforward way to find some covering set is to find all local cycles and choose one transition from each cycle. However, it is often the case that a cycle cannot occur without one of the other cycles occurring as well. Formally, a set of local cycles  $D$  covers a local cycle  $c$  iff for each global cycle  $o$  holds  $c \in P(o) \Rightarrow P(o) \cap D \neq \emptyset$ . A set of local cycles  $C$  is covered by  $D$  iff  $D$  covers each cycle in  $C$ . If some set of cycles  $D$  covers all local cycles, then it is sufficient to choose one transition from each cycle of  $D$ . In our studies of existing models, we have identified four common situations which can be used to identify when a cycle is covered by a set of other cycles:

- Let  $c$  be local cycle that at some point updates the value of a variable  $v$  to some value  $k'$  and later requires it to have another value  $k$  without updating  $v$  in the meantime. In order to be realisable, such cycles must be combined with a cycle which updates  $v$  to  $k$ . Let  $\text{need\_change\_to}(v, k)$  be the set of cycles which need  $v$  to be changed to  $k$  in order to be realisable. Let  $\text{changes\_to}(v, k)$  be the set of cycles which change  $v$  to  $k$ .
- A variation of this idea is based on cycles which either always increase or decrease a variable  $v$  (similar to a for-loop). These must be combined with cycles updating  $v$  in a non-increasing or non-decreasing manner, respectively. Let  $\text{increasing}(v)$  be the set of cycles which always increase  $v$  (and similarly for *decreasing*) and let  $\text{changes}(v)$  be the set of cycles which update  $v$ .
- Cycles synchronising via an action must obviously be paired. Let  $\text{sync}(a)$  be the set of cycles which synchronise on  $a$ .
- Cycles which at some point require a clock  $x$  to be bigger than  $k$  and later to be smaller than  $k$  without resetting  $x$  in the meantime must be combined with cycles resetting  $x$  (the opposite does not hold since clocks can be incremented by delaying). Let  $\text{needs\_reset}(x)$  be the set of cycles which need  $x$  to be reset and let  $\text{resets}(x)$  be the set of cycles which reset  $x$ .

**Lemma 2.** Let  $A_1 \parallel \dots \parallel A_n$ ,  $A_i = (L_i, l_i^0, E_i, \text{guard}_i, \text{sync}_i, \text{up}_i, I_i)$  be a network of timed automata. Let  $C_i$  be the set of local cycles of  $A_i$ , and  $C = \cup_i C_i$ . For

some  $i$ , let  $c \in C_i$ . Then  $D$  covers  $c$  if at least one of the following holds:

$$\begin{aligned} \exists v, k : c &\in \text{needs\_change\_to}(v, k) \wedge \text{changes\_to}(v, k) \subseteq D \\ \exists v : c &\in \text{increasing}(v) \wedge \text{changes}(v) \setminus \text{increasing}(v) \subseteq D \\ \exists v : c &\in \text{decreasing}(v) \wedge \text{changes}(v) \setminus \text{decreasing}(v) \subseteq D \\ \exists a : c &\in \text{sync}(a) \wedge \text{sync}(\bar{a}) \setminus C_i \subseteq D \\ \exists x : c &\in \text{need\_reset}(x) \wedge \text{resets}(x) \subseteq D \end{aligned}$$

Returning to our running example, we now see that  $\{A, B\}$  is covered by  $\{X, Y\}$  because the former is in  $\text{need\_change\_to}(i, 0)$  and the latter is the only cycle in  $\text{changes\_to}(i, 0)$ . Similarly,  $\{X, Y\}$  is covered by  $\{A, B, C\}$  because the former is in  $\text{sync}(a)$  and the latter is the only cycle in  $\text{sync}(\bar{a})$ .

All functions used in Lemma 2 may be obtained by static analysis, perhaps with the need of over-approximation *e.g.* in case an effect of a transition on a variable is found too complicated, the static analysis may simply suppose that the transition can change the variable to any value of its domain. The usefulness of the different heuristics depends on the type of models we want to analyse. For communication protocols using *need\\_change\\_to* seems sensible, whereas *increasing* and *decreasing* are much more useful in models containing **for** like constructs.

The choice of covering set determines the number of stored states. However, the number of stored states is not proportional to the size of the covering set, but rather to the frequency of the edges of the covering set in the symbolic state space (because we store targets of these edges). For example, in Fig. 1 the frequency of edge  $(Y, X)$  is  $4/12$ , whereas the frequency of edge  $(C, A)$  is  $2/12$ . The number of states stored for the covering set  $\{(Y, X)\}$  is indeed larger than for  $\{(C, A)\}$ . Unfortunately, it is very difficult to estimate the frequency of an edge from a static analysis. Therefore we propose to perform a *Random Walk Analysis*: Before the construction of the covering set we perform several random walks through the system and count the occurrences of edges. In this way we identify *random walk coefficients*. These are fairly good estimates of the frequencies. Thus we would like to find covering set with the smallest sum of random walk coefficients. However, since even finding the smallest covering set is as hard as reachability, we have to use a heuristic approach.

Figure 4 presents two possible algorithms for construction of a covering set. Different heuristics can be used for *choose\_cycle* and *choose\_transition* functions. It is advantageous to take these random walk coefficients into account in these heuristics. We refer to section 5.1 for a discussion of these heuristics and their comparison. Moreover, the ONE-PHASE-CONSTRUCTION algorithm can be used for verification of user-proposed sets. Note that this algorithm does need to construct all local cycles – in the worst case this can be exponential in the size of the automaton, although in practice the worst case is seldomly reached (for our industrial test cases it was maximally 500 cycles). If an automaton has a very large number of local cycles, it might be advantageous to find a set of covering transitions for this automaton using, for instance, depth first search, and then

**proc** TWO-PHASE-CONSTRUCTION

```

 $H = \text{covered} = \emptyset;$ 
while  $\text{covered} \neq C$  do
   $c = \text{choose\_cycle}(C \setminus \text{covered});$ 
   $H = H \cup \{c\};$ 
   $\text{covered} = \text{CLOSURE}(\text{covered} \cup \{c\});$ 
od
 $T = \emptyset;$ 
while  $H \neq \emptyset$  do
   $t = \text{choose\_transition}(H);$ 
   $T = T \cup \{t\};$ 
  foreach  $c \in H, t \in c$  do
     $H = H \setminus \{c\};$  od
od
return  $T;$ 
end

```

**proc** ONE-PHASE-CONSTRUCTION

```

 $H = T = \emptyset;$ 
while  $H \neq C$  do
   $t = \text{choose\_transition}(C \setminus H);$ 
   $T = T \cup \{t\};$ 
  foreach  $c \in C, t \in c$  do
     $H = H \cup \{c\};$  od
   $H = \text{CLOSURE}(H);$ 
od
return  $T;$ 
end

```

**Fig. 4.** Algorithms for computation of Covering set; CLOSURE( $H$ ) computes set of cycles already covered by  $H$  (using Lemma 2).

use the ONE-PHASE-CONSTRUCTION algorithm for the rest with this set as a starting set for  $T$ .

## 5 Experiments

A prototype of the techniques presented in this paper has been implemented in the real-time model-checker UPPAAL. We have performed exhaustive experiments with different heuristics for covering set construction, storing strategies (including different parameters), and combinations of strategies. Experiments were done on 12 different examples, including industrial case-studies and examples previously studied in the literature. Due to space considerations we only summarise the results and give the conclusions we have drawn from the experiments. The results reported in this section are based on the following examples (*Aut.* is the number of automata in the network; *Edges* is the number of edges):

Name	Aut.	Edges	Description
Fischer	4	20	Mutual exclusion protocol for 4 processes
Train-Gate	6	42	Train crossing with 4 trains
CSMA	7	58	CSMA/CD protocol for 6 processes
BRP	6	46	Bounded Retransmission Protocol
Dacapo	6	206	Start-up phase of a TDMA protocol
Token Ring	8	70	FDDI token ring for 7 stations
BOCDP	9	130	Bang & Olufsen Collision Detection Protocol
BOPDP	9	142	Bang & Olufsen Power Down Protocol
Buscoupler	16	198	Data link layer of ABB field bus protocol

In the following, the quality of different methods for the covering set construction is evaluated and then the proposed storing strategies are compared.

**Table 1.** Different heuristics for covering set construction; for each heuristic (O1 - T4) and model we report number of Edges (E) in the covering set, the sum of the random walk coefficients (RWC), and peak memory consumption of the reachability algorithm with this covering set.

	BRP			Train-gate			BOPDP			BOCDP		
	E	RWC	Memory	E	RWC	Memory	E	RWC	Memory	E	RWC	Memory
O1	2	0.15	31.5%	<b>2</b>	0.13	27.4%	<b>3</b>	0.2	18.3%	<b>5</b>	0.13	17.1%
O2	9	0.21	39.9%	<b>2</b>	0.12	30.8%	8	0.16	17.9%	18	0.15	22.5%
O3	11	0.21	39.9%	4	0.29	30.3%	12	0.16	17.9%	20	<b>0.08</b>	<b>13.3%</b>
O4	8	0.2	37.1%	5	0.23	28.8%	6	<b>0.14</b>	17.4%	14	0.25	29.9%
O5	<b>1</b>	<b>0.08</b>	<b>16.5%</b>	3	0.24	36.3%	<b>3</b>	0.26	33.8%	7	0.21	24.1%
T1	7	0.14	23.3%	<b>2</b>	0.12	30.8%	5	0.16	<b>9.3%</b>	10	0.18	24.9%
T2	10	0.14	27.1%	3	<b>0.11</b>	<b>25.1%</b>	10	0.15	11.3%	28	0.17	25.8%
T3	<b>1</b>	<b>0.08</b>	<b>16.5%</b>	17	0.53	51.5%	57	0.41	37.5%	53	0.37	48.8%
T4	<b>1</b>	<b>0.08</b>	<b>16.5%</b>	12	0.37	70.2%	22	0.4	46.8%	25	0.36	45.9%

Here *memory consumption* refers to the peak size of  $P \cup W$  during the computation, given as a percentage of all reachable states; *overhead* is the ratio between the number of visited states and the number of reachable states.

## 5.1 Covering Set Construction

At the moment, our prototype supports the heuristics based on *need\_change\_to* and *sync* for the identification of covering sets. Table 1 reports some representative results from using these heuristics. We have tried both one phase methods (O1-O5) and two phase methods (T1-T4) with different heuristics for the selection of a next cycle/transition, *e.g.* selection according to the number of newly covered cycles (O2, O3), the length of the cycle (T1, T2); random selection (O4); approach which chooses a suitable variable (channel) and then selects all cycles covering this variable (channel) (T3, T4); selecting min-RWC transition from max-RWC cycle (O1); selecting the transition with largest RWC(O5). The main observations from these experiments are the following:

- The number of edges that are needed to cover all cycles is quite small. From 12 tested models, some of them consisting of more than 10 automata and 150 edges, half of them can be covered by one or two edges and only one of them needs more than 5 edges.
- The sum of the random walk coefficients is a better static measure of the ‘quality’ of a covering set than the size of the set.
- None of the nine methods is ‘dominant’. The most ‘stable’ method seems to be one phase construction which selects next transition according to the ratio of newly covered cycles and random walk coefficient (O2).

Thus we conclude that the best approach is to have several different heuristics, construct several covering sets, and then choose the one with the smallest sum of random walk coefficients. This is the method that we have used in the following experiments.

## 5.2 Storing Strategies

The results from using different storing strategies are reported in Table 2. The 'entry point' strategy is a slightly improved<sup>1</sup> version of the strategy in [11], which is a special case of the covering set construction presented in this paper and which is already implemented in UPPAAL. The results for distance, random, and combination strategies depend on the choice of parameters. It is usually possible to achieve better results than those reported here by choosing parameters optimised for the given model. The parameters used in the table are those which give the best 'overall' results. We have made experiments with some other (combinations of these) strategies, but the results were usually similar to those presented.

For many of the examples, the combined strategy significantly reduces the amount of memory used (*e.g* for BRP, token ring, Train gate, Dacapo, BOCDP, BOPD and Buscoupler less than 20% of the state space needs to be represented at any time). In those cases the number of revisits is often relatively small (less than a factor of 2) and the runtime overhead is even smaller (the runtime overhead is bounded by the relative increase in the number of visited states). In fact, in some cases the reachability algorithm with reduction is even faster than the standard algorithm because a lot of time consuming operations are eliminated (since the passed list is smaller) and hence the exploration rate is higher.

**Table 2.** Comparison of strategies – the memory and overhead are reported.

	entry points	covering set	successors	random $p = 0.1$	distance $k = 10$	combination $k = 3$	
Fischer	27.1%	42.1%	47.9%	53.7%	67.6%	56.9%	
3,077	1.00	1.66	1.00	4.51	2.76	6.57	
BRP	70.5%	16.5%	19.8%	18.3%	15.8%	7.6%	
6,060	1.01	1.20	1.03	1.78	1.34	1.68	
Token Ring	33.0%	10.3%	20.7%	17.2%	17.5%	16.8%	
15,103	1.16	1.46	1.03	1.63	1.43	7.40	
Train-gate	71.1%	27.4%	24.2%	31.8%	24.2%	19.8%	
16,666	1.22	1.55	1.68	2.90	2.11	5.08	
Dacapo	29.4%	24.3%	24.9%	12.2%	12.7%	7.0%	
30,502	1.07	1.08	1.07	1.21	1.16	1.26	
CSMA	94.0%	75.9%	81.2%	105.9%	114.9%	120.3%	
47,857	1.06	2.62	1.40	7.66	2.83	6.82	
BOCDP	25.2%	22.5%	6.5%	10.2%	9.3%	4.5%	
203,557	1.00	1.01	1.08	1.02	1.01	1.09	
BOPDP	14.7%	13.2%	42.1%	15.2%	11%	4.3%	
1,013,072	2.40	1.33	1.02	1.52	1.14	1.74	
Buscoupler	53.2%	13.6%	40.5%	31.7%	24.6%	14.3%	
3,595,108	1.29	2.48	1.18	3.17	2.13	8.73	

The surprising fact is the small number of revisits. It is difficult to provide any theoretical explanation of this phenomenon (the worst-case behaviour is exponential). We just point out that the order in which vertices are visited becomes very important with reductions. Let us suppose that there are two

<sup>1</sup> The improved version exploits information about communication channels to a certain extent.

paths  $p_1$  and  $p_2$  from state  $s_1$  to state  $s_2$  and that state  $s_2$  is not going to be stored. Thus when the state is reached by the second path, it will be revisited. But if the paths have the same length (and this is very often the case) and we use breadth-first search order (*i.e.*  $W$  is implemented as queue), then  $s_2$  will still be in  $W$  when it is visited via the second path and thus it will not be revisited (at least not for now). With depth-first search order, it would be revisited anyway. We have verified this hypotheses on our examples. The overhead was significantly larger for depth-first search order than for breadth-first. This observation may suggest why the state-space caching in SPIN [6] was not very succesfull, because SPIN is DFS-based.

In two cases (Fischer's and CSMA), the combined strategy turns out to be less efficient than the existing entry point strategy used in UPPAAL. It turns out when using reductions not only the number of revisits increases, but also the peak size of the waiting list. In some cases, the size of the waiting list dominates the memory consumption. For example, the combination strategy in the CSMA example stores only 16% of the states in the passed list, but due to the large waiting list the peak memory consumption is much larger. We have observed this behaviour mainly for parametrised systems which consists of several copies of one process – these systems have a high degree of nondeterminism and interleaving and thus the waiting list grows faster. This problem can partially be solved by implementing the waiting list as a priority queue and giving priority to states which are not going to be stored (intuitively, we want to get rid of them as soon as possible). Since the order is not breadth-first the overhead increases considerably. We have performed experiments confirming this reasoning.

There is clearly a trade-off between space and time and also between passed list size and waiting list size. The combined strategy usually stores the least amount of states in the passed list (but due to the waiting list the actual memory consumption can be larger than for other strategies). The successor strategy is not the most memory efficient, but causes nearly no overhead in terms of revisited states.

## 6 Conclusion

The contributions of this paper include a number of complementary storing strategies as well as static techniques for determining small sets of covering transitions with the aid of a random walk analysis. Extensive experiments have been carried out with different heuristics for covering set construction, storing strategies and combinations of strategies. The experiments are very encouraging; on a variety of industrial case-studies the space-saving is more than 90% with only a moderate increase in runtime. Though the experiments have been conducted within the real time tool UPPAAL the strategies proposed are equally applicable to other model checkers (including finite-state ones). However, the strategies are particular useful for timed systems as partial order reduction has yet to be successfully developed in this context (*e.g.* see [4]).



## References

1. R. Alur and D. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.
2. G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In *Proc. of TACAS'03*, LNCS. Springer-Verlag, 2003.
3. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proc. of CAV'99*, LNCS. Springer-Verlag, 1999.
4. J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In *Proc. of CONCUR '98: Concurrency Theory*, 1998.
5. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking:  $10^{20}$  states and beyond. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1990.
6. P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.
7. P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proc. of IEEE Symp. on Logic in Computer Science*, pages 406–415, 1991.
8. R. M. Karp. Redcibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.
9. R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Statical partial order reduction. In Bernhard Steffen, editor, *Proc. of TACAS'98*, volume 1384 of LNCS, pages 345–357, Lisbon, Portugal, Mar.–Apr. 1998. Springer-Verlag.
10. F. Laroussinie and K. G. Larsen. Compositional Model Checking of Real Time Systems. In *Proc. of CONCUR '95*, LNCS. Springer-Verlag, 1995.
11. F. Larsson, K. G. Larsen, P. Pettersson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.
12. J. Lind-Nielsen, H. Reif Andersen, G. Behrmann, H. Hulgaard, K. J. Kristoffersen, and K. G. Larsen. Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. In Bernard Steffen, editor, *Proc. of TACAS'98*, number 1384 in LNCS, pages 201–216. Springer-Verlag, 1998.
13. A. Valmari. A Stubborn Attack on State Explosion. *Theoretical Computer Science*, 3, 1990.

# Calculating $\tau$ -Confluence Compositionally<sup>\*</sup>

Gordon J. Pace<sup>1</sup>, Frédéric Lang<sup>2</sup>, and Radu Mateescu<sup>2</sup>

<sup>1</sup> University of Malta (gordon.pace@um.edu.mt)

<sup>2</sup> INRIA Rhône-Alpes, France ({Frederic.Lang, Radu.Mateescu}@inrialpes.fr)

**Abstract.**  $\tau$ -confluence is a reduction technique used in enumerative model-checking of labeled transition systems to avoid the state explosion problem. In this paper, we propose a new on-the-fly algorithm to calculate partial  $\tau$ -confluence, and propose new techniques to do so on large systems in a compositional manner. Using information inherent in the way a large system is composed of smaller systems, we show how we can deduce partial  $\tau$ -confluence in a computationally cheap manner. Finally, these techniques are applied to a number of case studies, including the rel/REL atomic multicast protocol.

## 1 Introduction

An important area of research in model checking is the generation of restricted models using intuition and insight in the system in question to produce smaller state spaces — small enough to enumerate and manipulate. In practice, different techniques have been developed. Of interest to this paper, we note: *on-the-fly* model generation, where only the ‘interesting’ part of the model is generated; *partial-order reduction* [14] and the related  *$\tau$ -confluence* [8,17] reduction techniques which exploit independence of certain transitions in the system to discard unnecessary parts; and *compositional techniques* [7] where a model is decomposed into smaller parts, partially generated using knowledge about future interface components to avoid intermediate explosion.

In this paper we are mainly interested in deriving techniques which use structural information of the system to perform  $\tau$ -confluence reduction. Extracting general  $\tau$ -confluence of a flattened system can be costly and impractical. However, the user usually also provides the system in the form of a symbolic description, which we attempt to exploit at a low cost to calculate  $\tau$ -confluence. The information we use is the connection pattern of the network of communicating transition systems — composition expressions. At the leaves, we have transition system components, usually various magnitudes of size smaller than the whole system (especially if techniques such as projection [11] are first applied). Using the structure of the network, we can immediately deduce certain independence between transitions to be used for model reduction. We propose a new algorithm

---

<sup>\*</sup> This work has been partially supported by the European Research Consortium in Informatics and Mathematics (ERCIM).

to calculate partial  $\tau$ -confluence on-the-fly — similar in spirit to [2], but optimised in particular for flat transition systems. We then prove correct a number of laws which allow us to deduce  $\tau$ -confluence in a composition expression without the need of expensive calculations and show its performance when applied on a number of case studies, including the rel/REL atomic multicast protocol.

## 2 Related Work

An extensive and thorough study of  $\tau$ -confluence in process algebras and LTS verification can be found in [8]. In [17], the results are developed further, extending weak confluence conditions for divergent transition systems.

The ideas we develop in this paper heavily borrow from [9], in which a global (not on-the-fly) algorithm is given for calculating maximal  $\tau$ -confluence sets. The algorithmic complexity is of the order  $O(m * fanout_\tau^3)$ , where  $m$  is the number of transitions in the LTS, and  $fanout_\tau$  is the maximum number of  $\tau$  transitions exiting from a state. The paper also uses  $\tau$ -prioritisation and  $\tau$ -compression (where chains of  $\tau$  transitions are replaced by a single one), used to reduce a LTS, once a  $\tau$ -confluence set has been calculated. We use the same notion of  $\tau$ -confluence as in this paper mainly since discovering  $\tau$ -confluence sets under this definition is well-tractable. Our alternative algorithm to evaluate a maximal  $\tau$ -confluence set has complexity of the order  $O(m * fanout_\tau * fanout_\tau)$  and works on-the-fly. Furthermore, our  $\tau$ -confluence detection algorithm works equally well for LTSS which are divergent, and we use deduction to partially identify  $\tau$ -confluence in large systems by analysing their components.

[3,2] build upon the results of [17] and are closely related to our work, except that they concentrate on weak confluence. The algorithms work equally well with the stronger confluence condition we use. To calculate a  $\tau$ -confluent set, they use the symbolic description of the LTS (as guarded action/event systems) and feed conditions to an automated theorem prover to prove the independence of certain guards. In a certain sense, our algorithm to calculate the maximal  $\tau$ -confluent set can be seen as an extreme case of this approach — the LTS expanded to the actual description of the LTS transitions, and given the trivial nature of the resulting guards and transitions, we replace the automated theorem prover by a BES solver. Our symbolic description, based on composition expressions, differs from theirs, and allows for certain independence to be concluded easily, but does not allow symbolic reduction as is possible in their case.

$\tau$ -confluence is closely connected to partial-order reduction techniques [14]. The fact that  $\tau$  transitions are ‘partially’ invisible under branching and other weak bisimulations, means that independence of  $\tau$  transitions preserving bisimulation is possible, and can be useful in practice. In [16] is an analysis of partial-order methods applied to process algebras, that includes a set of conditions sufficient to guarantee branching bisimulation equivalence after reduction. As remarked in [2], these conditions are stronger than weak  $\tau$ -confluence. The conditions are not comparable to the notion of partial confluence we use, since we allow for confluence, but closing up to one step ahead. [16] allows for multiple

invisible transitions, but not for confluence. The conditions, however, closely relate to the conditions used in this and other  $\tau$ -confluence papers.

Several partial-order reduction techniques applied to compositions of LTSS have been proposed. Of interest are the  $\tau$ -diamond elimination technique presented in [4] (implemented for CSP in the FDR 2 tool) and a technique based on the detection of so-called  $\tau$ -inert transitions presented in [15] (implemented for CCS in the Concurrency Factory). Both consist in identifying  $\tau$ -transitions that do not need be interleaved with concurrent transitions, since the obtained behaviour would be equivalent (for some relation) to the one in which the  $\tau$ -transition is taken first. The difference relies on the properties being preserved under bisimulation in the case of behaviour equivalence preserved under reduction: weak bisimulation in the case of [15], and failure/divergence in the case of [4]), both of which do not preserve branching properties of the system. Additionally, our approach works on-the-fly, in combination with any verification tool of CADP, and for any language with a front-end for CADP.

### 3 Basic Definitions

**Definition:** A *labeled transition system* (LTS) is a quadruple  $\langle Q, Act, \rightarrow, q_0 \rangle$  where  $Q$  is the set of *states* of the system,  $Act$  is the set of possible *actions* the system may take (including a special invisible action  $\tau$ ),  $\rightarrow \subseteq Q \times Act \times Q$  is the set of *transitions* and  $q_0 \in Q$  is the *initial state* of the system.

Using standard conventions, we will write  $q \xrightarrow{a} q'$  to say that  $(q, a, q') \in \rightarrow$ , and for a set of actions  $G \subseteq Act$ ,  $\xrightarrow{G}$  is the transition relation  $\rightarrow$  restricted to actions in  $G$ .  $actions(q)$  is the set of actions possible from state  $q$ . If we may want to ‘ignore’ invisible transitions,  $q \xrightarrow{\bar{a}} q'$ , means that either  $q \xrightarrow{a} q'$ , or  $q = q'$  and  $a = \tau$  (note that this case does not necessarily imply that  $q \xrightarrow{\tau} q'$ ).  $\xrightarrow{\tau^*}$  is the reflexive transitive closure of  $\xrightarrow{\{\tau\}}$ . Finally, we say that an LTS is *divergent* if there exists an infinite sequence of states  $q_i$  such that for all  $i$ ,  $q_i \xrightarrow{\tau} q_{i+1}$ .

**Definition:** Given two LTSS  $S_1$  and  $S_2$  ( $S_i = \langle Q_i, Act, \rightarrow_i, q_{0i} \rangle$ ) a relation between the states of the two LTSS  $\simeq \subseteq Q_1 \times Q_2$  is said to be a *branching bisimulation* if for any  $q_1 \simeq q_2$ , the following two properties are satisfied:

1. for any  $q_1 \xrightarrow{a} q'_1$ , there exist  $q'_2, q''_2$  with  $q_2 \xrightarrow{\tau^*} q'_2 \xrightarrow{\bar{a}} q''_2$  and  $q_1 \simeq q'_2, q'_1 \simeq q''_2$ .
2. for any  $q_2 \xrightarrow{a} q'_2$ , there exist  $q'_1, q''_1$  with  $q_1 \xrightarrow{\tau^*} q'_1 \xrightarrow{\bar{a}} q''_1$  and  $q'_1 \simeq q_2, q''_1 \simeq q'_2$ .

The maximal branching bisimulation is a well-defined equivalence relation ( $\simeq_b$ ). We say that two LTSS are branching bisimilar ( $S_1 \simeq_b S_2$ ) if their initial states are branching bisimilar  $q_{01} \simeq_b q_{02}$ .

#### 3.1 $\tau$ -Confluence

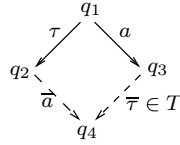
$\tau$ -confluence corresponds to the intuition that certain silent transitions do not change the set of transitions we can undertake now or in the future. If we can

calculate a set of silent transitions with this property, we can then reduce the LTS to obtain a smaller system.

Different levels of  $\tau$ -confluence have been defined in the literature. Some encompass more  $\tau$  transitions (and hence allow more powerful reductions), but are more expensive to calculate an appropriate confluent set. Others are more restrictive, but allow cheap  $\tau$ -confluence set deduction. In this paper we will concentrate on so-called *strong confluence* which we will refer to in the rest of the paper simply as confluence. The interested reader is referred to [8,17] for a whole hierarchy of  $\tau$ -confluence notions.

**Definition:** Given an LTS  $S = \langle Q, Act, \rightarrow, q_0 \rangle$ , and  $T \subseteq \{\tau\}$ , we say that  $T$  is  $\tau$ -confluent in  $S$  if: for every  $q_1 \xrightarrow{\tau} q_2 \in T$  and  $q_1 \xrightarrow{a} q_3$ , there exists a state  $q_4$  such that  $q_2 \xrightarrow{\bar{a}} q_4$  and  $q_3 \xrightarrow{\bar{\tau}} q_4 \in T$ .

The intuition is that every other outgoing transition of  $q_1$  can be emulated after the  $\tau$ -confluent transition. Graphically, the  $\tau$ -confluence can be seen in the following figures. Normal line transitions are given (universally quantified), whereas dashed transitions indicate that their existence must be proved:



**Proposition 1:** If  $q \xrightarrow{\tau} q'$  is a  $\tau$ -confluent transition in  $S$ , then  $q \simeq_b q'$ .

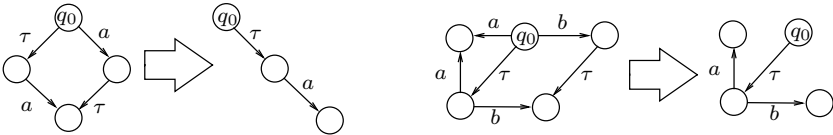
**Proposition 2:** The union of two  $\tau$ -confluent sets of an LTS  $S$  is itself a  $\tau$ -confluent set of  $S$ . We call the union of all  $\tau$ -confluent sets the *maximal  $\tau$ -confluent set*, and write it as  $\mathbb{T}(S)$ .

The proofs of these propositions can be found in [9].

### 3.2 Reduction Techniques

**Definition:** Given two LTSS  $S_1$  and  $S_2$  ( $S_i = \langle Q_i, Act, \rightarrow_i, q_{0i} \rangle$ ), we say that  $S_2$  is a  $\tau$ -prioritisation of  $S_1$  with respect to a  $\tau$ -confluent set  $T$ , if (i)  $\rightarrow_2 \subseteq \rightarrow_1$  and (ii) for every  $q \xrightarrow{a}_1 q'$ , either  $q \xrightarrow{a}_2 q'$  or for some  $q''$ ,  $q \xrightarrow{\tau}_2 q'' \in T$ .

The following figures show two examples of  $\tau$ -prioritisation (with unreachable states removed):



**Proposition 3:** If  $S_1$  is a  $\tau$ -prioritisation of a non-divergent LTS  $S_2$  with respect to  $T$ , then  $S_1 \simeq_b S_2$ .

The proof can be found in [9].  $\tau$ -prioritisation thus allows reduction of non-divergent systems with respect to a  $\tau$ -confluent set, maintaining equivalence modulo branching bisimulation. The main problem with  $\tau$ -prioritisation is that

it is restricted to non-divergent systems. However, one can augment the prioritisation to calculate and eliminate on-the-fly  $\tau$  cycles. Alternatively, other reduction techniques [3,13] have been defined in the literature (see Section 2) and can be used.

## 4 Calculating $\tau$ -Confluence Using Boolean Equations

In this section, we present an on-the-fly algorithm to calculate the maximal  $\tau$ -confluent set.

Definitional boolean equation systems without negation are a well-known and studied field. The following is a short resumé of definitions and results to set the picture for the translation algorithm we propose for on-the-fly  $\tau$ -confluence calculation.

### 4.1 Boolean Equation Systems

**Definition:** A *boolean equation system* (BES) is a set of variables  $V$  split into two disjoint subparts  $V_d$  and  $V_c$ , with their definition  $\delta \in V \rightarrow 2^V$ . Variables in  $V_d$  are seen as defined in terms of a disjunction over the definition set, while those in  $V_c$  as a conjunction:

**Definition:** An *interpretation*  $I$  of a BES is a subset of variables  $V$  of the equation system,  $I \subseteq V$ . Variable  $v$  is said to be *satisfied* in  $I$  if  $v \in I$ . An interpretation is said to be *valid* if the definition function holds:

$$(\forall v : V_d . \delta(v) \cap I \neq \emptyset) \wedge (\forall v : V_c . \delta(v) \subseteq I)$$

**Proposition 4:** The union of all valid interpretations of a BES  $Eq$  is itself a valid interpretation. This is called the *greatest fixed point solution*:  $(\nu V. Eq)$ .

Standard algorithms exist to evaluate the greatest fixed point of a boolean equation system. In particular, we are mainly interested in an on-the-fly algorithm — a local one resolving only the necessary variables we may require. Such algorithms can be found in [12,1] and work in both a breadth-first and depth-first fashion. This problem can be solved in time proportional to the number of variables and the size of the definition sets.

### 4.2 Translating $\tau$ -Confluence of LTSS into Boolean Equations

It is rather straightforward to translate the definition of  $\tau$ -confluence in Section 3.1 into a BES whose validity implies the confluence of individual  $\tau$  transitions.

**Definition:** Given an LTS  $S$ , we introduce a conjunctive variable for every  $\tau$  transition, and a disjunctive variable for every half-terminated diamond in the  $\tau$ -confluence diagram:

$$V_c \stackrel{df}{=} \{c_{q_1, q_2} \mid q_1 \xrightarrow{\tau} q_2\}, \quad V_d \stackrel{df}{=} \{d_{q_1, q_2, q_3}^a \mid q_1 \xrightarrow{\tau} q_2, q_1 \xrightarrow{a} q_3\}$$

The intuitive interpretation we will use is that (i) every confluent  $\tau$  transition has to be able to close *all* half-diamonds (conjunction) and (ii) every half-diamond has to be closed by *some* other confluent  $\tau$  transition (disjunction).

The boolean variables  $c_{q_1, q_2}$  will be satisfiable if and only if  $q_1 \xrightarrow{\tau} q_2$  is confluent, while  $d_{q_1, q_2, q_3}^a$  is satisfiable if and only if the half-diamond can be satisfactorily closed.

**Conjunctive variables:**  $c_{q_1, q_2}$  should be satisfiable if and only if all extended half-diamonds which are not trivially closed (via a direct  $a$  transition from  $q_2$  to  $q_3$ ) can be closed:

$$\delta(c_{q_1, q_2}) \stackrel{df}{=} \{d_{q_1, q_2, q_3}^a \mid q_1 \xrightarrow{\tau} q_2, q_1 \xrightarrow{a} q_3, q_2 \not\xrightarrow{q} q_3\}$$

**Disjunctive variables:**  $d_{q_1, q_2, q_3}^a$  is satisfiable if and only if there is some  $\tau$  transition from  $q_3$  to some  $q_4$  which (i) closes the diamond, and (ii) is  $\tau$ -confluent:

$$\begin{aligned} \delta(d_{q_1, q_2, q_3}^a) &\stackrel{df}{=} \{c_{q_3, q_4} \mid q_2 \xrightarrow{a} q_4, q_3 \xrightarrow{\tau} q_4\} \\ \delta(d_{q_1, q_2, q_3}^\tau) &\stackrel{df}{=} \{c_{q_3, q_4} \mid q_2 \xrightarrow{\tau} q_4, q_3 \xrightarrow{\tau} q_4\} \cup \{c_{q_3, q_2} \mid q_3 \xrightarrow{\tau} q_2\} \end{aligned}$$

Note that in the case of  $a = \tau$ , the diamond may be closed as a triangle (see the figures depicting how  $\tau$ -confluence diagrams can be closed.)

**Proposition 5:** The translation is sound and complete: Given a valid interpretation  $I$  of a translated LTS  $S$ ,  $\{q_1 \xrightarrow{\tau} q_2 \mid c_{q_1, q_2} \in I\}$  is a  $\tau$ -confluent set (soundness), and for any  $\tau$ -confluent set  $T$ , there is a valid interpretation  $I$  such that  $I \cap V_c = \{c_{q_1, q_2} \mid q_1 \xrightarrow{\tau} q_2 \in T\}$  (completeness).

From this proposition, it then follows that:

**Theorem 1:** Calculating the greatest fixed point of the BES obtained by translating an LTS gives the maximal  $\tau$ -confluent set.

### 4.3 Complexity

Consider variables  $V_c$ . We have  $m_\tau$  (the number of  $\tau$  transitions) such variables. Furthermore, the definition set of each variable is bounded above by  $fanout * fanout_\tau$  ( $fanout$  is the maximum number of successors of a state in the LTS,  $fanout_\tau$  is the maximum number of  $\tau$ -successors). Now consider the disjunctive variables  $V_d$ . We have  $m_\tau * fanout$  such variables (for each  $\tau$  transition, we have an entry for each other transition which can be taken from the source node). The definition sets of these variables never exceeds  $fanout_\tau$  entries.

Recall that a BES can be solved in time proportional to the number of variables plus the size of the definition sets. The complexity of resolving  $\tau$ -confluence using our algorithm is thus  $O(m_\tau fanout * fanout_\tau)$ . This compares favourably with the algorithm given in [9] which has complexity  $O(m_\tau * fanout_\tau^3)$ .

However, this is pessimistic view of the complexity. Due to the regular nature of the equations (conjunctions of disjunctions), and the fact that we also know that the disjunctive variables are never reused (a disjunctive variable is revisited only through a conjunctive one), we can hone the algorithm to work more efficiently (for example, by not caching disjunctive variables).

## 5 Definitions and Basic Results: Composition Expressions

**Definition:** A *composition expression* is an object in the abstract type *Exp*:

$$Exp ::= LTS \mid \text{hide } G \text{ in } Exp \mid Exp \parallel_G Exp$$

The basic building blocks are LTSS, together with the hiding operator (re-names any label in the action set  $G$  to  $\tau$ ) and synchronous composition (actions in  $G$  are synchronised, the rest can happen independently). One can add other operators to this family, but these usually suffice for a decomposed view of a system.

A composition expression describes the way a family of LTSS communicate together, but can be seen itself as an LTS.

**Hiding:** In  $(\text{hide } G \text{ in } E_1)$ , if  $E_1$  is the LTS  $\langle Q_1, Act_1, \rightarrow_1, q_{01} \rangle$ , the resultant LTS is  $\langle Q_1, Act_1, \rightarrow, q_{01} \rangle$  where  $\rightarrow$  is the smallest relation generated by the following structured operational semantics rules:

$$\frac{q_1 \xrightarrow{a}_1 q'_1, a \notin G}{q_1 \xrightarrow{a} q'_1} \qquad \frac{q_1 \xrightarrow{a}_1 q'_1, a \in G}{q_1 \xrightarrow{\tau} q'_1}$$

**Synchronous composition:** In the composition expression  $(E_1 \parallel_G E_2)$ , if  $E_i$  corresponds to the LTS  $\langle Q_i, Act_i, \rightarrow_i, q_{0i} \rangle$ , the resultant LTS is  $\langle Q_1 \times Q_2, Act_1 \cup Act_2, \rightarrow, (q_{01}, q_{02}) \rangle$  where  $\rightarrow$  is the smallest relation generated by the following structured operational semantics rules:

$$\frac{q_1 \xrightarrow{a}_1 q'_1, a \notin G}{(q_1, q_2) \xrightarrow{a} (q'_1, q_2)} \quad \frac{q_2 \xrightarrow{a}_2 q'_2, a \notin G}{(q_1, q_2) \xrightarrow{a} (q_1, q'_2)} \quad \frac{q_1 \xrightarrow{a}_1 q'_1, q_2 \xrightarrow{a}_2 q'_2, a \in G}{(q_1, q_2) \xrightarrow{a} (q'_1, q'_2)}$$

For the sake of brevity, in contexts where we speak of expressions, unless otherwise stated, the LTS generated by expression  $E$  will be  $\langle Q, Act, \rightarrow, q_0 \rangle$ , and that of expression  $E_i$  will be  $\langle Q_i, Act_i, \rightarrow_i, q_{0i} \rangle$ .

**Definition:** The subterm relation over composition expressions  $\sqsubseteq$  is defined to be the reflexive, transitive closure of the smallest relation  $\sqsubset_1$  satisfying:

$$E \sqsubset_1 \text{hide } G \text{ in } E, E \sqsubset_1 E \parallel_G E', E \sqsubset_1 E' \parallel_G E$$

We say that a transition  $q \xrightarrow{a} q'$  of  $E$  is *immediately generated from* a transition  $q_1 \xrightarrow{a_1}_1 q'_1$  of  $E_1$  ( $E_1 \sqsubset_1 E$ ) if the derivation of the former transition using the operational semantic rules requires the use of the latter. Thus, for example,  $q_1 \xrightarrow{\tau} q_2$  in  $(\text{hide } a \text{ in } E)$  is immediately generated from  $q_1 \xrightarrow{a} q_2$  in  $E$ .

We are mainly interested in the transitive closure of this relation:  $\uparrow_{E_1}^{E_2} \sqsubseteq \rightarrow_1 \times \rightarrow_2$  (where  $E_1 \sqsubseteq E_2$ ), which relates transitions in  $\rightarrow_2$  (of  $E_2$ ) with the transitions in  $\rightarrow_1$  (of  $E_1$ ) contributing to their generation.

For this definition to make sense, we will make the simplifying assumption that an expression will not contain common subexpressions (all the leaf LTSS are different). This is done to simplify the presentation but can be easily remedied either by tagging different leaf nodes (different tag for every leaf) or by reasoning in terms of expression contexts.



The decomposition law states that if  $E_1 \sqsubseteq E_2 \sqsubseteq E_3$ :  $\uparrow_{E_3}^{E_2} \circ \uparrow_{E_2}^{E_1} = \uparrow_{E_3}^{E_1}$  (where  $r \circ s$  is the relation composition of  $r$  and  $s$ ).

Similarly, we can talk about a transition generating another:  $t \downarrow_{E_2}^{E_1} t'$  means that  $t$  is a *generator* of  $t'$ .  $\downarrow_{E_2}^{E_1}$  is simply the inverse of  $\uparrow_{E_2}^{E_1}$ .

We define relation application as usual:  $R(X) \stackrel{df}{=} \{y \mid \exists x \in X . x R y\}$ .

**Definition:** Given a composition expression  $E$ , the actions hidden above, and synchronised above a subexpression  $E_1$  are defined as:

$$\begin{aligned} Hidden_E(E_1) &\stackrel{df}{=} \bigcup \{G \mid \text{hide } G \text{ in } E_2 \sqsubseteq E, E_1 \sqsubseteq E_2\} \\ Synchronised_E(E_1) &\stackrel{df}{=} \bigcup \{G \mid E_2 \parallel_G E_3 \sqsubseteq E, E_1 \sqsubseteq E_2 \vee E_1 \sqsubseteq E_3\} \end{aligned}$$

Given  $E_1 \sqsubseteq E$ , we define  $Tau_E(E_1)$  to be the set of labels such that transitions in  $E_1$  whose label appears in  $Tau_E(E_1)$  are guaranteed to be transformed into  $\tau$  transitions in  $E$ :

$$Tau_E(E_1) \stackrel{df}{=} Hidden_E(E_1) \setminus Synchronised_E(E_1)$$

**Proposition 6:** Given  $E_1 \sqsubseteq E$ , every transition labelled by  $Tau_E(E_1)$  generates at least one  $\tau$  transition, and nothing but  $\tau$  transitions:

$$\forall t : \xrightarrow{Tau_E(E_1)} . \quad \uparrow_{E_1}^E(t) \neq \emptyset \wedge \uparrow_{E_1}^E(t) \subseteq \{\tau\}$$

**Proof:** The proof follows from structural induction with the inductive hypothesis that in expression  $E_2$  ( $E_1 \sqsubseteq E_2 \sqsubseteq E$ ), a non-empty set of  $\{\tau\} \cup Tau_E(E_1)$  transitions generates a non-empty set of  $\{\tau\} \cup Tau_E(E_2)$  transitions.

Furthermore, since by definition,  $Tau_E(E) = \emptyset$ , the conclusion follows.  $\square$

We will write the expression obtained by replacing in  $E$  the occurrence of sub-expression  $E_2$  by  $E_1$  as  $E[E_1/E_2]$ .

**Proposition 7:** Branching bisimilarity is preserved in composition expressions: If  $E_1 \sqsubseteq E$  and  $E_1 \simeq_b E_2$  then  $E \simeq_b E[E_2/E_1]$ .

**Proposition 8:** Actions in  $Tau_E(E_1)$  can be hidden immediately in  $E_1$ . Given  $E_1 \sqsubseteq E$  and  $G \subseteq Tau_E(E_1)$ :  $E[\text{hide } G \text{ in } E_2/E_1] \simeq_b E[E_2/E_1]$ .

Consider a  $\tau$  transition in a leaf LTS, which is not confluent. Just by looking at the leaf in question, we can sometimes deduce that the transition can never become confluent. Transitions about which we cannot guarantee this will be called *potential*  $\tau$ -confluent transitions. We identify a set of transitions which we will later prove that all  $\tau$  transitions generated higher up in the expression tree, will be generated by transitions in this set.

The intuition is the following: a transition is potentially confluent if (i) either it is already invisible, or its action will be hidden higher up in the expression tree, (ii) hidden, it satisfies the  $\tau$ -confluence conditions on all other outgoing transitions except (iii) it may not satisfy the  $\tau$ -confluence conditions with respect to transitions which may later disappear (synchronised above).

**Definition:** Given  $E_1 \sqsubseteq E$ ,  $P_1 \subseteq \xrightarrow{G}_1$  (where  $G = Hidden_E(E_1) \cup \{\tau\}$ ) is said to be a *potential*  $\tau$ -confluence set if, for all  $q_1 \xrightarrow{a}_1 q_2 \in P_1$  and  $q_1 \xrightarrow{b}_1 q_3$  with  $b \notin Synchronised_E(E_1)$ , then either  $q_3 \xrightarrow{a?}_1 q_2 \in P_1$  or there exists  $q_4$  such that

$q_3 \xrightarrow{a?}_1 q_4 \in P_1$  and  $q_2 \xrightarrow{b?}_1 q_4$ .  $q \xrightarrow{a?} q'$  is defined as  $q \xrightarrow{a} q' \vee (a \in G \wedge \exists a' \in G . q \xrightarrow{a'} q')$ .

**Proposition 9:** The union of all potential  $\tau$ -confluence sets of  $E_1$  with respect to  $E$  ( $E_1 \sqsubseteq E$ ) is itself a potential  $\tau$ -confluence set. We call this the *maximal potential  $\tau$ -confluence set* and write it as  $\mathbb{P}_E(E_1)$ .

**Proposition 10:** If  $T$  is a  $\tau$ -confluent set of  $E_1$  ( $E_1 \sqsubseteq E$ ),  $T$  is also a potential  $\tau$ -confluence set of  $E_1$  with respect to  $E$ .

**Proof:** Consider  $q_1 \xrightarrow{\tau} q_2 \in T$ . Since it is a  $\tau$ -confluent transition, for any  $q_1 \xrightarrow{a} q_3$ , there exists  $q_4$  such that  $q_2 \xrightarrow{\bar{a}} q_4$  and  $q_3 \xrightarrow{\bar{\tau}} q_4 \in T$ . Consider the different cases from the  $\bar{a}$  and  $\bar{\tau}$ : (i)  $a = \tau$ ,  $q_2 = q_4$ ,  $q_3 = q_4$  (ii)  $a = \tau$ ,  $q_2 = q_4$ ,  $q_3 \xrightarrow{\tau} q_2 \in T$  (iii)  $q_2 \xrightarrow{a} q_4$ ,  $q_3 = q_4$  (iv)  $q_2 \xrightarrow{a} q_4$ ,  $q_3 \xrightarrow{\tau} q_4 \in T$ . These satisfy the property required of potential  $\tau$ -confluence.  $T$  is thus a potential  $\tau$ -confluence set.  $\square$

**Proposition 11:** If  $E_1 \sqsubseteq E$ , then  $\mathbb{T}(E_1) \subseteq \mathbb{P}_E(E_1)$ .

**Proof:** The proof follows immediately from propositions 2, 9 and 10.  $\square$

## 6 Calculating $\tau$ -Confluence in Composition Expressions

We now give a number of results to deduce  $\tau$ -confluence in composition expressions without applying the algorithm on the top-level LTS, which can be very large.

### 6.1 Discovering $\tau$ -Confluence in Composition Expressions

The basic result we will apply to reduce composition expressions, is that  $\tau$ -confluent transitions can only generate  $\tau$ -confluent transitions. This can be very useful, especially if the leaf LTSS are reduced using  $\tau$ -prioritisation, where in the resultant LTS, the  $\tau$ -confluent transitions become the only transitions leaving a state, making them trivially recognisable as  $\tau$ -confluent ones.

**Theorem 2:** If  $T_1$  is a  $\tau$ -confluent transition set of  $E_1$  ( $E_1 \sqsubseteq E$ ) then  $\uparrow_{E_1}^E(T_1)$ , the set of transitions of  $E$  generated from  $T_1$ , is a  $\tau$ -confluent transition set of  $E$ .

This theorem together with the reduction techniques given in Section 3 provides us with two approaches to reduce an LTS in a compositional manner. One way is to calculate and label confluent transitions in the leaves, and use this information to deduce a confluence set in the top level LTS and perform reduction on-the-fly as the top level LTS is generated (using either  $\tau$ -prioritisation or any other technique). Another approach is to reduce the leaves using maximal  $\tau$ -prioritisation (leaving only one confluent outgoing transition, when one is available), thus making sure that as the top level LTS is generated, confluent transitions in the leaves are easily recognisable (unique  $\tau$  transitions leaving a state) and use this information to generate the reduced LTS. The latter has the advantage that confluence information needs not be stored.

## 6.2 Doing More than $\tau$ Transitions

One way in which new  $\tau$ -confluence can manifest itself is via new  $\tau$  transitions appearing from the *hide* operator. In general, we cannot just treat transitions which are eventually hidden as invisible transitions, because if they are synchronised before being hidden, they may disappear due to the other branch not complementing the required transition. In the case of hidden transitions which are not synchronised, we can either push the *hide* operator into the expression to generate  $\tau$  transitions as early as possible, or treat them as invisible transitions (despite the fact that they are not  $\tau$  transitions). The second solution is preferable, since it does not destroy the structure of the expression as given by the user, and avoids adding new expression nodes, resulting in slower analysis. The following result justifies their treatment analogous to  $\tau$  transitions.

**Theorem 3:** Given  $E_1 \sqsubseteq E$  and  $T_1 \subseteq \xrightarrow{1}_{\tau}^{Tau_E(E_1)}$  which satisfies the confluence conditions if replaced by  $\tau$  transitions, and  $E_2$ , a  $\tau$ -prioritisation of  $E_1$  with respect to  $T_1$ , then  $E[E_2/E_1] \simeq_b E$ .

## 6.3 Some $\tau$ Transitions Are Not Worth the Bother

Finally, we can not only identify transitions which are, and will remain confluent, but also ones which can under no circumstances become confluent. Since within composition expressions we can only partially identify  $\tau$ -confluent transitions, we may want to apply  $\tau$ -confluence algorithm at the top-most level once again. If certain transitions can be identified as certainly not being  $\tau$ -confluent during the expression tree traversal, we can apply the  $\tau$ -confluence detection algorithm on a smaller set of transitions. The main theorem in this section allows us to do precisely this by using the notion of potential  $\tau$ -confluence.

**Lemma 1:** If  $P_2$  is a potential  $\tau$ -confluent set of  $E_2$  with respect to  $E$  ( $E_1 \sqsubseteq E_2 \sqsubseteq E$ ) then  $\downarrow_{E_1}^{E_2}(P_2)$  is a potential  $\tau$ -confluent set of  $E_1$  with respect to  $E$ .

**Lemma 2:** If  $E_1 \sqsubseteq E_2 \sqsubseteq E$ , then  $\mathbb{P}_E(E_2) \subseteq \uparrow_{E_1}^E(\mathbb{P}_E(E_1))$

**Theorem 4:** Some transitions need never be checked for confluence. If  $E_1 \sqsubseteq E$ :

$$\mathbb{T}(E) \cap (\rightarrow_1 \setminus \uparrow_{E_1}^E(\mathbb{P}_E(E_1))) = \emptyset$$

**Proof:** From lemma 2 and proposition 10 we can now conclude that:

$$\mathbb{T}(E) \subseteq \uparrow_{E_1}^E(\mathbb{P}_E(E_1))$$

from which the theorem directly follows.  $\square$

Thus, by identifying and marking the complement of the maximal potential  $\tau$ -confluent set in the leaf nodes, we can mark transitions which they generate at higher levels in the expression tree. Using this theorem, we are guaranteed that these transitions are not confluent, and we can thus reduce the computation required to identify a  $\tau$ -confluent set of the LTS generated by the whole composition expression.

## 7 Tools and Applications

We have implemented the techniques described within the CADP toolkit [10] in the OPEN/CÆSAR environment. A collection of front-ends enable the compilation of source languages into C code which includes a function to access the LTS described by the system which is explored on-the-fly by the verification back-ends. EXP.OPEN is a front-end for composition expressions, while CÆSAR.OPEN is a front-end for the LOTOS language and GENERATOR is a back-end that explicitly generates the reachable state space of a system.

A variant of GENERATOR, named  $\tau$ -CONFLUENCE, detects and prioritises  $\tau$ -confluent transitions on-the-fly, using Boolean Equation Systems. EXP.OPEN has been extended to enable  $\tau$ -confluence detection (**branching** option), by taking an account of the composition expression as stated in Theorems 2 and 3. More precisely, in global LTS of a composition expression  $E$ , EXP.OPEN prioritises the transitions that were detected as  $\tau$ -confluent in the components of  $E$ . Additionally, some locally visible transitions are also prioritised, knowing that they will lead to  $\tau$ -confluent transitions in the global LTS of  $E$ .

EXP.OPEN flattens the composition expression into a tuple of LTSS and a set of so-called *synchronization vectors*. If  $n$  is the size of the LTS tuple, each synchronization vector is a tuple of size  $n + 1$ , whose elements are either labels or a special *null* value. The first  $n$  elements represent labels of transitions that must be fireable from the corresponding LTS current state components (none if element is null), whereas the last element (which must not be null) is the label of the resulting transition in the produced LTS. Working globally on the expression also allows us to identify certain locally confluent transitions which do not fall under the framework proposed in this paper. More details will be found in a related technical report. EXP.OPEN also calculates transitive closures of  $\tau$ -confluent transitions (to avoid entering circuits of  $\tau$ -confluent transitions), and hence compresses successive  $\tau$ -confluent transitions into a single one.

These tools have been used to generate the state space of the rel/REL protocol previously studied in [5,11]. The rel/REL protocol is an atomic multicast protocol between a transmitter and several receivers. This protocol is *reliable* in the sense that it allows arbitrary failures of the stations involved in the communication. The protocol guarantees the following two properties: (1) when a message  $M$  is sent by the transmitter, either every functioning station correctly receives  $M$ , or  $M$  is not received by any of the stations, and (2) messages are received in the same order as they are sent. Two underlying assumptions are needed to guarantee correctness: that crashed stations stop sending and receiving messages, and that functioning stations can always communicate with each other. The overall compositional structure of the system with two receivers is given by the following composition expression:

```
hide R_T1, R_T2, R1, R2, DEPOSE1, DEPOSE2 in
  CRASH_TRANSMITTER || {R_T1, R_T2} (
    (RECEIVER_THREAD1 || {R_T1, R1, R2, GET, CRASH, DEPOSE1} FAIL_RECEIVER1)
    || {R1, R2}
    (RECEIVER_THREAD2 || {R_T2, R1, R2, GET, CRASH, DEPOSE2} FAIL_RECEIVER2) )
```

**Table 1.** (a) Leaf LTS sizes using normal and  $\tau$ -prioritised generation, (b) Cost of normal and  $\tau$ -prioritised composition expression generation.

	Normal		$\tau$ -prioritised		Difference %	
	states	transitions	states	transitions	states	transitions
CRASH_TRANSMITTER	85	108	73	84	14%	22%
RECEIVER_THREAD $n$	16 260	167 829	16 260	115 697	0%	31%
FAIL_RECEIVER $n$	130	1 059	130	1 059	0%	0%

	Normal	$\tau$ -prioritised	Difference %
Number of states	249 357	114 621	54%
Number of transitions	783 470	220 754	72%
EXP.OPEN execution time	2'23"	2'10"	9%
EXP.OPEN memory consumption (Kb)	5 776	3 944	32%
SVL execution time	3'05"	3'03"	1%

The composition of LTSS **RECEIVER\_THREAD $n$**  and **FAIL\_RECEIVER $n$**  ( $n = 1, 2$ ) defines the behaviour of receiver  $n$ , including the possibility of a crash. The LTS **CRASH\_TRANSMITTER** describes the behaviour of the transmitter. These LTSS are generated from a LOTOS description of the system, detailed in [5].

In our experiments, performed using SVL scripts [6], we have compared two state-space generation approaches for the rel/REL protocol: (i) *Normal generation*: leaf LTSS and composition expression are generated normally, without optimisation (using respectively **CÆSAR.OPEN/GENERATOR** and **EXP.OPEN/GENERATOR**). (ii)  *$\tau$ -prioritised generation*: leaf LTSS are generated using the **CÆSAR.OPEN/ $\tau$ -CONFLUENCE** tools and composition expression is generated using **EXP.OPEN branching** together with **GENERATOR**.

Experiment results are displayed in Table 1. From these results,  $\tau$ -prioritisation techniques on composition expressions seem very promising. Various reasons contribute to the success of  $\tau$ -prioritisation. Although both **FAIL\_RECEIVERS** are purely sequential, **RECEIVER\_THREADS** and **CRASH\_TRANSMITTER** use parallel composition of processes performing silent transitions. This generates many  $\tau$ -confluent transitions, which are detected by  $\tau$ -confluence. Also, as a consequence of successful  $\tau$ -prioritisation in three of the five leaves of the composition expression, **EXP.OPEN** avoids the creation of new  $\tau$ -confluent diamonds. Additionally, a lot of transitions present in leaves are hidden at the top-level of the composition expression, some of which are confluent.

Note that applying  $\tau$ -prioritisation at the top-level gives no further reduction showing that we have identified the maximal  $\tau$ -confluent set.

To see what gain can be obtained on examples less adapted with respect to these observations, we have applied the  $\tau$ -confluence technique to systems with purely sequential leaf components. We have chosen examples from the CADP distribution: two versions of the Alternating Bit Protocol and five versions of a Distributed Leader Election Protocol. Table 2 shows the results. Note that in this case, comparing execution times is irrelevant, since  $\tau$ -prioritisation of sequential

**Table 2.** Difference ratios for several case studies

Difference %	EXP.OPEN		State Space	
	time	memory	states	trans
Alternating Bit(1)	9%	0%	4%	25%
Alternating Bit(2)	-4%	0%	6%	27%
Distributed Leader Election(1)	-57%	3%	11%	24%
Distributed Leader Election(2)	-21%	0%	12%	23%
Distributed Leader Election(3)	-88%	5%	5%	11%
Distributed Leader Election(4)	-90%	-1%	0%	8%
Distributed Leader Election(5)	-102%	-1%	0%	0%

components is known to be useless. It is very encouraging to note that in all experiments, the overhead in memory consumption is negligible, since memory more than time is usually the bottleneck in verification.

## 8 Conclusions

$\tau$ -confluence can be an effective technique to reduce transition systems at a reasonable cost. We propose to use composition expressions to help identify independent transitions resulting in  $\tau$ -confluence at a negligible cost. One question arising is whether we can do better by enriching the set of composition operators.

In this paper we concentrate on results for strong confluence, mainly because we have no efficient way of recognising weak confluence at the leaf nodes. However, it would be useful to extend these results, especially since certain leaf nodes may be small enough to calculate larger sets of more weakly confluent transitions. Overall, we believe that composition structure information can, in various contexts, be used to improve existing algorithms.

## References

1. H.R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994.
2. S. Blom and J. van de Pol. State space reduction by proving confluence. In *Computer Aided Verification 2002*, number 2404 in LNCS, 2002.
3. S.C.C. Blom. Partial  $\tau$ -confluence for efficient state space generation. Technical Report SEN-R0123, CWI, Amsterdam, 2001.
4. A.W. Roscoe et al. Hierarchical compression for model-checking CSP or how to check  $10^{20}$  dining philosophers for deadlock. In *TACAS'95*, number NS-95-2, 1995.
5. Jean-Claude Fernandez, Hubert Garavel, Laurent Mounier, Anne Rasse, Carlos Rodriguez, and Joseph Sifakis. A toolbox for the verification of LOTOS programs. In *International Conference on Software Engineering*, pages 246–259, 1992.
6. Hubert Garavel and Frédéric Lang. SVL: a scripting language for compositional verification. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 377–392. Kluwer Academic Publishers, 2001.

7. Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *Computer Aided Verification*, volume 531 of *LNCS*, 1990.
8. J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1–2):47–81, 15 December 1996.
9. J. F. Groote and J. van de Pol. State space reduction using partial tau-confluence. In *Mathematical Foundations of Computer Science*, number 1893 in *LNCS*, 2000.
10. J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, M. Sighireanu. CADP: a protocol validation and verification toolbox. *CAV'96*, *LNCS* 1102, 1996.
11. Jean-Pierre Krimm and Laurent Mounier. Compositional state space generation from LOTOS programs. In *Proceedings of TACAS'97*, volume 1217 of *LNCS*, 1997.
12. Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*. to appear.
13. R. Nalumasu and G. Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3), May 2002.
14. D.A. Peled, V.R. Pratt, and G.J. Holzmann, editors. *Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.
15. Y.S. Ramakrishna and S.A. Smolka. Partial-order reduction in the weak modal mu-calculus. In *Proceedings of CONCUR'97*, volume 1243 of *LNCS*, 1997.
16. A. Valmari. Stubborn set methods for process algebras. In *Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.
17. Mingsheng Ying. Weak confluence and  $\tau$ -inertness. *Theoretical Computer Science*, 238(1–2):465–475, May 2000.

# Author Index

- Abdulla, Parosh Aziz, 236  
Abu-Haimed, Husam, 407  
Alur, Rajeev, 67  
Armoni, Roy, 368  
Ashar, Pranav, 206
- Bardin, Sébastien, 118  
Bartzis, Constantinos, 249  
Behrmann, Gerd, 433  
Berezin, Sergey, 407  
Beyer, Dirk, 122  
Boigelot, Bernard, 193, 223  
Bordini, Rafael H., 110  
Bouyer, Patricia, 180  
Bozga, Liana, 219  
Bryant, Randal E., 141, 154, 341  
Bultan, Tevfik, 249
- Camphenhout, David Van, 27  
Chakravorty, Gaurav, 167  
Chechik, Marsha, 210  
Ciardo, Gianfranco, 40  
Clarke, Edmund, 126  
Cleaveland, Rance, 106  
Colón, Michael A., 420  
Cook, Byron, 141
- D'Souza, Deepak, 180  
Dang, Zhe, 93  
Dill, David L., 407  
Dong, Yifei, 215  
Drusinsky, Doron, 114
- Eisner, Cindy, 27
- Finkel, Alain, 118  
Fisher, Michael, 110  
Fisman, Dana, 27  
Fix, Limor, 368  
Flaisher, Alon, 368  
Flanagan, Cormac, 355
- Ganai, Malay, 206  
Geilen, Marc, 394  
Glusman, Marcelo, 328  
Grumberg, Orna, 54, 126, 275, 368
- Gupta, Aarti, 206  
Gurfinkel, Arie, 210
- Havlicek, John, 27  
Henzinger, Thomas A., 262  
Herbreteau, Frédéric, 193  
Heyman, Tamir, 54  
Hungar, Hardi, 315
- Ibarra, Oscar H., 93
- Jhala, Ranjit, 262  
Jodogne, Sébastien, 193  
Jonsson, Bengt, 236  
Joshi, Rajeev, 355
- Katz, Shmuel, 328  
Kesten, Yonit, 381
- Lahiri, Shuvendu K., 141, 341  
Lakhnech, Yassine, 219  
Lang, Frédéric, 446  
Larsen, Kim G., 433  
Legay, Axel, 223  
Leroux, Jérôme, 118  
Leuştean, Laurenţiu, 301  
Lewerentz, Claus, 122  
Lustig, Yoad, 27
- Madhusudan, P., 67, 180  
Majumdar, Rupak, 262  
Mateescu, Radu, 446  
McIsaac, Anthony, 27  
McMillan, K.L., 1  
Moura, Leonardo de, 14
- Namjoshi, Kedar S., 288  
Niese, Oliver, 315  
Nilsson, Marcus, 236  
Noack, Andreas, 122
- Obdržálek, Jan, 80  
d'Orso, Julien, 236  
Ou, Xinming, 355
- Pace, Gordon J., 446  
Pandya, Paritosh K., 167



Pardavila, Carmen, 110  
 Pelánek, Radek, 433  
 Périn, Michaël, 219  
 Petit, Antoine, 180  
 Petrucci, Laure, 118  
 Piterman, Nir, 368, 381  
 Pnueli, Amir, 381

Qadeer, Shaz, 262

Ramakrishnan, C.R., 215  
 Roşu, Grigore, 301  
 Rueß, Harald, 14

San Pietro, Pierluigi, 93  
 Sankaranarayanan, Sriram, 420  
 Saxe, James B., 355  
 Schuster, Assaf, 54  
 Sengupta, Bikram, 106  
 Seshia, Sanjit A., 154  
 Shoham, Sharon, 275  
 Siminiceanu, Radu, 40

Sipma, Henny B., 420  
 Smolka, Scott A., 215  
 Sorea, Maria, 14  
 Steffen, Bernhard, 315

Talupur, Muralidhar, 126  
 Tiemeyer, Andreas, 368  
 Torre, Salvatore La, 67

Vardi, Moshe Y., 368  
 Venkatesan, Ram Prasad, 301  
 Visser, Willem, 110

Wang, Chao, 206  
 Wang, Dong, 126  
 Whittle, Jon, 301  
 Wolper, Pierre, 223  
 Wooldridge, Michael, 110

Xie, Gaoyan, 93

Yang, Zijiang, 206