



# Formal Methods in Computer Aided Design

Austin, Texas  
11–14 November 2007

*2007 Formal Methods in Computer Aided Design*

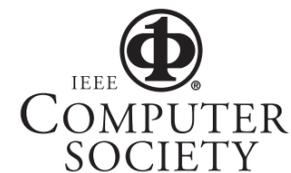


Published by the IEEE Computer Society  
10662 Los Vaqueros Circle  
P.O. Box 3014  
Los Alamitos, CA 90720-1314

IEEE Computer Society Order Number P3023  
Library of Congress Number 2007935533  
ISBN 0-7695-3023-0



 IEEE Computer Society



Sponsored by  
FMCAD Inc.  
IEEE Council on Electronic Design Automation (CEDA)



*Proceedings*

---

Formal Methods in  
Computer Aided Design

FMCAD 2007



*Proceedings*

---

Formal Methods  
in Computer Aided Design

FMCAD 2007

November 11–14, 2007  
Austin, Texas, USA



Los Alamitos, California  
Washington • Tokyo



All rights reserved.

*Copyright and Reprint Permissions:* Abstracting is permitted with credit to the source. Libraries may photocopy beyond the limits of US copyright law, for private use of patrons, those articles in this volume that carry a code at the bottom of the first page, provided that the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Other copying, reprint, or republication requests should be addressed to: IEEE Copyrights Manager, IEEE Service Center, 445 Hoes Lane, P.O. Box 133, Piscataway, NJ 08855-1331.

*The papers in this book comprise the proceedings of the meeting mentioned on the cover and title page. They reflect the authors' opinions and, in the interests of timely dissemination, are published as presented and without change. Their inclusion in this publication does not necessarily constitute endorsement by the editors, the IEEE Computer Society, or the Institute of Electrical and Electronics Engineers, Inc.*

IEEE Computer Society Order Number P3023

ISBN 0-7695-3023-0

ISBN 978-0-7695-3023-9

Library of Congress Number 2007935533

*Additional copies may be ordered from:*

IEEE Computer Society  
Customer Service Center  
10662 Los Vaqueros Circle  
P.O. Box 3014  
Los Alamitos, CA 90720-1314  
Tel: + 1 800 272 6657  
Fax: + 1 714 821 4641  
<http://computer.org/cspress>  
[csbooks@computer.org](mailto:csbooks@computer.org)

IEEE Service Center  
445 Hoes Lane  
P.O. Box 1331  
Piscataway, NJ 08855-1331  
Tel: + 1 732 981 0060  
Fax: + 1 732 981 9667  
[http://shop.ieee.org/store/  
customer-service@ieee.org](http://shop.ieee.org/store/customer-service@ieee.org)

IEEE Computer Society  
Asia/Pacific Office  
Watanabe Bldg., 1-4-2  
Minami-Aoyama  
Minato-ku, Tokyo 107-0062  
JAPAN  
Tel: + 81 3 3408 3118  
Fax: + 81 3 3408 3553  
[tokyo.ofc@computer.org](mailto:tokyo.ofc@computer.org)

*Individual paper REPRINTS may be ordered at: <[reprints@computer.org](mailto:reprints@computer.org)>*

Editorial production by Lisa O'Conner

Cover art production by Joe Daigle/Studio Productions

Printed in the United States of America by Applied Digital Imaging



***Conference Publishing Services***

<http://www.computer.org/proceedings/>

# Table of Contents

FMCAD 2007

## Formal Methods in Computer Aided Design

<b>Preface</b> .....	xx
<b>Organizing Committee</b> .....	xxi
<b>Program Committee</b> .....	xix
<b>Referees</b> .....	xxiv

---

### SAT-Based Methods

Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC.....	3
<i>Jocelyn Simmonds, Jessica Davies, Arie Gurfinkel, and Marsha Chechik</i>	
Improved Design Debugging using Maximum Satisfiability .....	13
<i>Sean Safarpour, Mark Liffiton, Hratch Mangassarian, Andreas Veneris, and Karem Sakallah</i>	
Industrial Strength SAT-based Alignability Algorithm for Hardware Equivalence Verification .....	20
<i>Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili</i>	
Boosting Verification by Automatic Tuning of Decision Procedures .....	27
<i>Frank Hutter, Domagoj Babic, Holger Hoos, and Alan Hu</i>	

### High-Level System Analysis

Verifying Correctness of Transactional Memories .....	37
<i>Ariel Cohen, John O'Leary, Amir Pnueli, Mark Tuttle, and Lenore Zuck</i>	
Algorithmic Analysis of Piecewise FIFO Systems .....	45
<i>Naghmeh Ghafari, Arie Gurfinkel, Nils Klarlund, and Richard Trefler</i>	
Transaction Based Modeling and Verification of Hardware Protocol Implementations.....	53
<i>Xiaofang Chen, Steven German, and Ganesh Gopalakrishnan</i>	
Automating Hazard Checking in Transaction-Level Microarchitecture Models.....	62
<i>Yogesh Mahajan and Sharad Malik</i>	

## Abstraction-Based Methods

Computing Abstractions by integrating BDDs and SMT .....	69
<i>Roberto Cavada, Alessandro Cimatti, Anders Franzen, Kalyanasundaram Krishnamani, Marco Roveri, and R.K. Shyamasundar</i>	
Induction in CEGAR for Detecting Counterexamples.....	77
<i>Chao Wang, Aarti Gupta, and Franjo Ivancic</i>	
Lifting Propositional Interpolants to the Word-Level .....	85
<i>Daniel Kroening and Georg Weissenbacher</i>	

## Software Analysis Methods

Global Optimization of Compositional Systems .....	93
<i>Fadi Zaraket, John Pape, Adnan Aziz, Margarida Jacome, and Sarfraz Khurshid</i>	
Cross-Entropy Based Testing .....	101
<i>Hana Chockler, Benny Godlin, Eitan Farchi, and Sergey Novikov</i>	

## Symbolic Trajectory Evaluation

Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation .....	111
<i>Yan Chen, Yujing He, Fei Xie, and Jin Yang</i>	
A Logic for GSTE .....	119
<i>Edward Smith</i>	
Automatic Abstraction in Symbolic Trajectory Evaluation.....	127
<i>Sara Adams, Magnus Bjork, Tom Melham, and Carl-Johan Seger</i>	

## Specification Theory

A Coverage Analysis for Safety Property Lists .....	139
<i>Koen Claessen</i>	
What triggers a behavior?.....	146
<i>Orna Kupferman and Yoad Lustig</i>	
Two-Dimensional Regular Expressions for Compositional Bus Protocols .....	154
<i>Kathi Fisler</i>	
A Quantitative Completeness Analysis for Property-Sets.....	158
<i>Martin Oberkönig, Martin Schickel, and Hans Ekeking</i>	

## **Industrial-Strength Verification**

Automated Extraction of Inductive Invariants to Aid Model Checking.....	165
<i>Michael Case, Alan Mishchenko and Robert Brayton</i>	
Checking Safety by Inductive Generalization of Counterexamples to Induction .....	173
<i>Aaron Bradley and Zohar Manna</i>	
Fast Minimum Register Retiming Via Binary Maximum-Flow.....	181
<i>Aaron Hurst, Alan Mishchenko, and Robert Brayton</i>	
Formal Verification of Partial Good Self-Test Fencing Structures .....	188
<i>Adrian Seigler, Gary Van Huben, and Hari Mony</i>	
Case study: Integrating FV and DV within the Verification of Intel® Core(TM) Microprocessor .....	192
<i>Alon Flaisher, Alon Gluska, and Eli Singerman</i>	

## **Reasoning about Physical Systems**

Circuit-Level Verification of a High-Speed Toggle .....	199
<i>Chao Yan and Mark R. Greenstreet</i>	
Combining Symbolic Simulation and Interval Arithmetic for the Verification of AMS Designs .....	207
<i>Mohamed Zaki, Ghiath Al Sammane, Sofiene Tahar, and Guy Bois</i>	
Analyzing Gene Relationships for Down Syndrome with Labeled Transitions Graphs.....	216
<i>Neha Rungta, Hyrum Carroll, Eric Mercer, Randall Roper, Mark Clement, and Quinn Snell</i>	

## **Advanced Theorem-Proving Applications**

A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware.....	223
<i>Julien Schmaltz</i>	
Modeling Time-Triggered Protocols and Verifying their Real-Time Schedules .....	231
<i>Lee Pike</i>	
A Mechanized Refinement Framework for Analysis of Custom Memories .....	239
<i>Sandip Ray and Jayanta Bhadra</i>	

<b>Author Index</b> .....	243
---------------------------	-----

# Preface

This volume contains the proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD07), held in Austin, Texas, USA on November 11-14, 2007. FMCAD 2007 is the seventh in a series of conferences on the theory and applications of formal methods in hardware and system verification. FMCAD provides a leading forum to researchers and practitioners in academia and industry for presenting and discussing groundbreaking methods, technologies, theoretical results, and tools for reasoning formally about computing systems. FMCAD covers formal aspects of computer-aided system design including verification, specification, synthesis, and testing. In the past, FMCAD was held in the United States on even years and its sister conference, Correct Hardware Design and Verification Methods (CHARME), was held in Europe on odd years. Recently, these two conferences decided to merge to form an annual conference with a unified international community. As a result, FMCAD has become a yearly conference.

FMCAD 2007 received 65 regular paper submissions (of 80 submitted abstracts) and 17 short paper submissions. Each paper was reviewed by at least four Program Committee members, with the assistance of external sub-reviewers. The Program Committee finally accepted a total of 23 regular papers and 8 short papers. In addition to the technical papers, the conference program included a pre-conference day of invited tutorials, two invited talks, and two panel sessions. The invited tutorials were:

- Robert Brayton (UC Berkeley), “The Synergy between Logic Synthesis and Equivalence Checking”
- Randal E. Bryant (CMU), “Modeling Data in Formal Verification: Bits, Bit Vectors, or Words”
- Niklas Eén (Cadence), “Practical SAT”
- Farid Najm (U Toronto), “Power Management for VLSI Circuits and the Need for High-Level Power Modeling and Design Exploration”

The invited talks were:

- Wolfgang Kunz (U Kaiserslautern), “Formal Verification of Systems-on-Chip — Industrial Experiences and Scientific Perspectives”
- Eli Singerman (Intel), “Verification of Embedded Software in Industrial Microprocessors”

The first panel was entitled “Formal Verification: A Business Perspective” and was moderated by Aarti Gupta (NEC). It included presentations by a variety of EDA and semiconductor companies, including Cadence, Jasper Design Automation, Mentor Graphics, and Synopsys. The second panel was entitled “FMCAD2027: Will the ‘FM’ Have a Real Impact on the ‘CAD?’” and was moderated by William Joyner (SRC). It included presentations by Robert Jones (Intel), Andreas Kuehlmann (Cadence), and Carl Pixley (Synopsys).

We would like to acknowledge the support of our generous sponsors: FMCAD Inc., the IEEE Council on Electronic Design Automation (CEDA), Cadence, Freescale Semiconductor Inc., IBM Corporation, Intel Corporation, Jasper Design Automation, Mentor Graphics, NEC Labs America, and Synopsys. We also would like to thank our Panel Chairs: William Joyner (SRC) and Aarti Gupta (NEC); our Tutorials Chair: Natasha Sharygina (U Lugano); our Publicity Chair: Alper Sen (Freescale); our Webmasters: Sandip Ray (UT) and Hari Mony (IBM); and our Local Arrangement Chair: Andy Martin (IBM). We additionally thank Andrei Voronkov for assistance with the EasyChair system, and Igor Markov for assistance in using the Duplication Detection (DUDE) system. We acknowledge the Program Committee and their sub-referees for their diligent efforts in reviewing these papers. We finally would like to thank the FMCAD Steering Committee for their guidance.

**The FMCAD Chairs,**  
Jason Baumgartner (IBM)  
Mary Sheeran (Chalmers)

# Organizing Committee

## **Chairs**

Jason Baumgartner, *IBM Corporation, USA*

Mary Sheeran, *Chalmers University of Technology, Sweden*

## **Benchmarks**

Panagiotis Manolios, *Georgia Institute of Technology, USA*

## **Local Arrangements**

Andy Martin, *IBM Corporation, USA*

## **Panels**

Aarti Gupta, *NEC Laboratories America, USA*

William Joyner, *Semiconductor Research Corporation, USA*

## **Publicity**

Alper Sen, *Freescale Semiconductor Inc., USA*

## **Tutorials**

Natasha Sharygina, *University of Lugano, Switzerland*

## **Webmasters**

Hari Mony, *IBM Corporation, USA*

Sandip Ray, *University of Texas, USA*

# Program Committee

Mark Aagaard, *University of Waterloo, Canada*  
Jason Baumgartner, *IBM Corporation, USA*  
Armin Biere, *Johannes Kepler University, Austria*  
Per Bjesse, *Synopsys, USA*  
Dominique Borrione, *Grenoble University, France*  
Gianpiero Cabodi, *Politecnico di Torino, Italy*  
Alessandro Cimatti, *ITC-irst, Trento, Italy*  
Koen Claessen, *Chalmers University of Technology, Sweden*  
Cindy Eisner, *IBM Haifa Research Laboratory, Israel*  
Steven German, *IBM Research Division, USA*  
Ganesh Gopalakrishnan, *University of Utah, USA*  
Aarti Gupta, *NEC Laboratories America, USA*  
Alan J. Hu, *University of British Columbia, Canada*  
Warren Hunt, *University of Texas, USA*  
Steven Johnson, *Indiana University, USA*  
Robert Jones, *Intel Corp., USA*  
Daniel Kroening, *ETH Zurich, Switzerland*  
Andreas Kuehlmann, *Cadence Laboratories, USA*  
Wolfgang Kunz, *University of Kaiserslautern, Germany*  
Jeremy Levitt, *Mentor Graphics, USA*  
Panagiotis Manolios, *Georgia Institute of Technology, USA*  
Andy Martin, *IBM Research Division, USA*  
Tom Melham, *Oxford University, UK*  
Alan Mishchenko, *University of California at Berkeley, USA*  
Ken McMillan, *Cadence Labs, USA*  
John O'Leary, *Intel Corp., USA*  
Wolfgang Paul, *Saarland University, Germany*  
Carl Pixley, *Synopsys, USA*  
Natasha Sharygina, *University of Lugano, Switzerland*  
Mary Sheeran, *Chalmers University of Technology, Sweden*  
Anna Slobodova, *Intel Corp., USA*  
Richard Trefler, *University of Waterloo, Canada*

# Referees

Christoph Albrecht  
Tamarah Arons  
Gadiel Auerbach  
Adnan Aziz  
Gérard Basler  
Shoham Ben-David  
Sergey Berezin  
Jesse Bingham  
Magnus Björk  
Nicolas Blanc  
Karl Brace  
Angelo Brillout  
Roberto Bruttomesso  
Annette Bunker  
Michael Case  
Donald Chai  
Satrajit Chatterjee  
Yury Chebiryak  
Xiaofang Chen  
Hana Chockler  
Ching-Tsun Chou  
Vlad Ciubotariu  
Byron Cook  
Vijay D'Silva  
Jared Davis  
Niklas Eén  
John Erickson  
Dana Fisman  
Anders Franzén  
Oded Fuhrmann  
Danny Geist  
Naghme Ghafari  
Alberto Griggio  
Jim Grundy  
John Harrison

Keijo Heljanko  
Pei-Hsin Ho  
Himanshu Jain  
Barbara Jobstmann  
Toni Jussila  
Roope Kaivola  
Rouwaida Kanj  
Dmitry Korchemny  
Sava Krstić  
Robert Krug  
Jim Kukula  
Zarrin Langari  
Tim Leonard  
Guodong Li  
Sacha Loitz  
Jean Christophe Madre  
Freddy Mang  
Eric Mercer  
In-Ho Moon  
Shiri Moran  
Katell Morin-Allory  
Marco Murciano  
Richard Mushlin  
Chris Meyers  
Serita Nelesen  
Ziv Nevo  
Minh Nguyen  
Sergio Nocco  
Sergey Novikov  
Avigail Orni  
Evgeny Pavlenko  
Edgar Pek  
Laurence Pierre  
Mitra Purandare  
Stefano Quer

David Rager  
Fahim Rahim  
Sandip Ray  
Erik Reeber  
Marco Roveri  
Sitvanit Ruah  
Joseph N. Ruskiewicz  
Smruti Sarangi  
Viktor Schuppan  
Ohad Shacham  
Hazem Shehata  
Eli Singerman  
Sudarshan Srinivasan  
Christian Stangier  
Sol Swords  
Murali Talupur  
Max Thalmaier  
Saurabh Tiwary  
Stefano Tonetta  
Stavros Tripakis  
Aliaksei Tsitovich  
Rachel Tzoref  
Sarvani Vakkalanka  
Helmut Veith  
Vinod Viswanath  
Daron Vroon  
Chao Wang  
Markus Wedler  
Georg Weissenbacher  
Christoph M. Wintersteiger  
Jessie Xu  
Yu Yang  
Karen Yorav  
Emmanuel Zarpas  
Qiang

# Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC

Jocelyn Simmonds\*, Jessica Davies\*, Arie Gurfinkel<sup>†</sup> and Marsha Chechik\*

\*University of Toronto, Toronto, ON M5S 3G4, Canada.

Email: {jsimmond,jdavies,chechik}@cs.toronto.edu

<sup>†</sup>SEI at Carnegie Mellon University, Pittsburgh, PA 15213-2612, USA.

Email: arie@sei.cmu.edu

**Abstract**—When model-checking reports that a property holds on a model, *vacuity detection* increases user confidence in this result by checking that the property is satisfied in the intended way. While vacuity detection is effective, it is a relatively expensive technique requiring many additional model-checking runs. We address the problem of efficient vacuity detection for Bounded Model Checking (BMC) of LTL properties, presenting three partial vacuity detection methods based on the efficient analysis of the resolution proof produced by a successful BMC run. In particular, we define a characteristic of resolution proofs – *peripherality* – and prove that if a variable is a source of vacuity, then there exists a resolution proof in which this variable is peripheral. Our vacuity detection tool, *VaqTree*, uses these methods to detect vacuous variables, decreasing the total number of model-checking runs required to detect all sources of vacuity.

## I. INTRODUCTION

Model-checking [1] is a widely-used automated technique for verification of both hardware and software artifacts that checks whether a temporal logic property is satisfied by a finite-state model of the artifact. If the model does not satisfy the property, a counterexample, which can aid in debugging, is produced. If the model *does* satisfy the property, no information about why it does so is provided by the model-checker alone. A positive answer without any additional information can be misleading, since a property may be satisfied in a way that was not intended. For instance, a property “every request is eventually acknowledged” is satisfied in an environment that never generates requests.

*Vacuity detection* [2]–[5] is an automatic sanity check that can be applied after a positive model-checking run in order to gain confidence that the model and the property capture the desired behaviours. Informally, a property is said to be vacuous if it has a subformula which is not relevant to its satisfaction, or if the property itself is a tautology. Conversely, a property is satisfied non-vacuously if every part of the formula is important – even a slight change to the formula affects its satisfaction.

In this paper, we focus on vacuity detection for SAT-based Bounded Model Checking (BMC). Given a BMC problem with a particular bound, we wish to determine if the property holds vacuously on the model up to this bound. In this context, a naive method for detecting vacuity is to replace subformulas of the temporal logic property with unconstrained boolean variables and run BMC for each such substitution. If the property with some substitution still holds on the model, the property is

vacuous. This naive approach is expensive, since in the worst case it requires as many model-checking runs as there are subformulas in the property. Our goal is to reduce the number of model-checking runs required to detect vacuity. We do this by detecting some vacuity through novel and inexpensive techniques reported in this paper, and complete the method by running the naive algorithm on the remaining atomic subformulas. The key to our technique is that SAT-based BMC can automatically provide useful information (a resolution proof) beyond a decision whether the property holds on the model; we exploit such proofs for partial vacuity detection.

In SAT-based BMC, the property and the behavior of the model are encoded in a propositional theory, such that the theory is satisfiable if and only if the formula does not hold. When the property does hold, a DPLL-based SAT solver can produce a resolution proof that derives false from a subset of the clauses in the theory called the UNSAT core. Intuitively, the resolution proof provides an explanation why the property is not falsified by the model, and the UNSAT core determines the relevant parts of the model and the property [6].

In this paper, we develop three methods of increasing precision (*irrelevance*, *local irrelevance* and *peripherality*) to analyze the resolution proof to achieve partial vacuity detection. These algorithms are used by our vacuity detection tool, *VaqTree*, in order to reduce the number of model-checking runs required to find all sources of vacuity, thus reducing execution times. Irrelevance and local irrelevance detect vacuity based on which variables appear in the UNSAT core, and in which locations. However, as these methods only examine the UNSAT core, their precision is limited. The peripherality algorithm examines the *structure* of the resolution proof, identifying as vacuous those variables that are not necessary or central to the derivation of false. This method is as precise as can be achieved through analyzing a single resolution proof, and its running time is linear in the size of the resolution proof and the number of variables in the property. Our experience shows that local irrelevance is the ideal candidate for replacing naive vacuity.

The remainder of the paper is organized as follows. Sec. II presents some required background, followed, in Sec. III by our definition of vacuity, the naive algorithm for LTL vacuity detection using BMC, and an overview of work in the vacuity detection field. Sec. IV presents the three algorithms that detect vacuity by analyzing a resolution proof. Our experimental

results are presented in Sec. V. We conclude with a summary, additional related work, and suggestions for future work in Sec. VI.

## II. BACKGROUND

In this section, we review bounded model-checking and resolution proofs.

### A. Bounded Model-Checking

*Bounded model-checking* (BMC) [7] is a method for determining whether a linear temporal logic (LTL) formula  $\varphi$  holds on a finite state system represented by a Kripke structure  $K$  up to a finite number of steps. An instance of a BMC problem, denoted by  $BMC_k(K, \varphi)$ , is whether  $K \models_k \varphi$ , where  $\models_k$  is the  $k$ -depth satisfaction relation. An informal description of LTL formulas, Kripke structures and BMC is given in [8], and detailed definitions can be found in [1], [7].

To determine whether  $K \models_k \varphi$ , the problem is converted to a propositional formula  $\Phi$  (see [7], [9], [10]) which is satisfiable if and only if there exists a length- $k$  counterexample to  $K \models_k \varphi$ .  $\Phi$  is then given to a SAT solver which decides its satisfiability. The propositional encoding represents the behavior of  $K$  up to  $k$  steps with a *path constraint*  $CL_K$ , and encodes all counterexamples to  $\varphi$  of length  $k$  in an *error constraint*  $CL_e$ . Therefore, if the theory  $CL_K \cup CL_e$  is satisfiable, there is a path through  $K$  which obeys the transition relation and falsifies  $\varphi$ . The value of each variable  $v$  of  $K$  at each time step is represented using new boolean variables  $v_i$  ( $0 \leq i \leq k$ ), called *timed variables*.

The transition relation can be represented symbolically by a propositional formula over the variables  $V$  and primed variables  $V'$  (which represent the variables in the next state). For example, in the model in Fig. 1(a), the transition relation is represented by the formula  $R = (p \wedge \neg q \wedge \neg p' \wedge q') \vee (\neg p \wedge q \wedge \neg p' \wedge q')$ . The path constraint is obtained by substituting the timed variables  $V_i$  for  $V$  in  $R$ , and replacing  $V'$  by the timed variables for the next step,  $V_{i+1}$ . This is repeated for each  $0 \leq i < k$ , and the resulting propositional formulas are conjoined along with a formula representing the initial state [7]. In Fig. 1(a), if  $k = 1$ ,

$$CL_K = (p_0 \wedge \neg q_0) \wedge ((p_0 \wedge \neg q_0 \wedge \neg p_1 \wedge q_1) \vee (\neg p_0 \wedge q_0 \wedge \neg p_1 \wedge q_1))$$

$CL_e$  is encoded according to a recursive procedure which removes the temporal and logical operators from the property [7], e.g., the algorithm encodes  $\varphi = \mathbf{G}p$ , where  $p$  is a propositional variable, expanded up to  $k = 2$ , by the formula  $\neg p_0 \vee \neg p_1 \vee \neg p_2$ .

After the boolean formulas for the path and error constraints are calculated, they are converted to *Conjunctive Normal Form* (CNF) before being passed to a SAT solver. If the solver reports that  $CL_K \cup CL_e$  is unsatisfiable, it means that there is no length- $k$  counterexample to  $\varphi$ ; otherwise, a satisfying assignment is returned. When a DPLL-based SAT solver processes an unsatisfiable theory, a resolution derivation of false (or the empty clause) is implicitly constructed [11], [12]. This resolution proof is used to verify that false can indeed be derived from  $CL_K \cup CL_e$  [13].

### B. Resolution Proofs

Resolution is an inference rule that is applied to propositional clauses to produce logical consequences. A *clause* is a disjunction of boolean variables and their negations. For example,  $(v_1 \vee \neg v_2 \vee v_5)$  is a clause stating that at least one of  $v_1, \neg v_2$  or  $v_5$  must be true. The resolution rule takes two clauses, where one contains a variable  $v$  and the other – its negation  $\neg v$ , and produces a clause containing the union of the two clauses minus  $v$  and  $\neg v$ . For example, resolving  $(v_1 \vee \neg v_2 \vee v_5)$  and  $(v_2 \vee v_6)$  produces the *resolvent*  $(v_1 \vee v_5 \vee v_6)$ .

A *resolution proof*  $\Pi$  is a directed acyclic graph whose nodes are labeled by propositional clauses.  $\Pi$  represents a tree of resolutions between the clauses labeling its nodes. Its *roots* are the nodes with no parents; otherwise, all nodes have exactly two parents. The nodes with no children are called the *leaves*. For example, the roots of resolution proof  $\Pi$  in Fig. 1(b) are  $Roots(\Pi) = \{(\neg r_0), (r_0 \vee p_0), (\neg p_0 \vee q_0), (\neg p_0 \vee \neg q_0), (p_0)\}$ , and the leaf of  $\Pi$  is the empty clause, i.e.,  $Leaf(\Pi) = \text{false}$ . Given a non-root node labeled by the clause  $c$ , and the labels of its parents,  $c_1$  and  $c_2$ ,  $c$  is the resolvent since it has been produced by resolving  $c_1$  and  $c_2$  on some variable  $v$ . A resolution proof  $\Pi$  is a *proof of unsatisfiability* of a set of clauses  $A$  if and only if all roots of  $\Pi$  belong to  $A$ , and one of the leaves of  $\Pi$  is the empty clause. For example, Fig. 1(b) shows a resolution proof of the unsatisfiability of  $Roots(\Pi)$ . If a propositional theory in CNF is unsatisfiable, an *UNSAT core* is an unsatisfiable subset of its clauses.

Given two disjoint sets of clauses  $A$  and  $B$ , a variable  $v$  is said to be *local* to  $A$  if and only if  $v$  appears in  $A$  but does not appear in  $B$ , and  $v$  is said to be *global* if it appears in both  $A$  and  $B$ . In Fig. 1(b), if  $Roots(\Pi) = A \cup B$ , where  $A = \{(\neg r_0), (r_0 \vee p_0), (\neg p_0 \vee q_0)\}$  and  $B = \{(\neg p_0 \vee \neg q_0), (p_0)\}$ , then  $r_0$  is local to  $A$ , and the rest are global.

## III. DEFINING VACUITY

This paper uses the following definition of vacuity.

**Definition 1** *Let  $K$  be a Kripke structure,  $\varphi$  be a formula satisfied by  $K$  (i.e.,  $K \models \varphi$ ), and  $p$  be a variable. Then,  $\varphi$  is  $p$ -vacuous in  $K$  iff  $\varphi[p \leftarrow x]$  is satisfied by  $K$ , where  $x$  is a variable not occurring in  $K$  or in  $\varphi$ .*

We use  $\varphi[p \leftarrow x]$  to indicate that all occurrences of  $p$  in  $\varphi$  are replaced by  $x$ .

Similarly, it is possible to define vacuity in the BMC setting.

**Definition 2** *Let  $K$  be a Kripke structure,  $\varphi$  be a formula s.t.  $K \models_k \varphi$ , and  $p$  be a variable.  $\varphi$  is  $k$ -step  $p$ -vacuous iff  $K \models_k \varphi[p \leftarrow x]$ , where  $x$  is a variable not occurring in  $K$  or in  $\varphi$ .*

If  $\varphi$  is  $k$ -step  $p$ -vacuous, we call  $p$  a  *$k$ -step vacuous variable*. A property  $\varphi$  is  *$k$ -step vacuous* if and only if  $\varphi$  contains a  $k$ -step vacuous variable. Therefore, our techniques aim to find the  $k$ -step vacuous variables of  $\varphi$ . The qualifier “ $k$ -step” is omitted in the remainder of the paper but should be understood implicitly in the BMC context.

In the remainder of the paper, we avoid referring to  $k$ -vacuity, focusing instead on those variables  $p$  that are used to prove that a property is  $k$ -vacuous. When we say that a property  $\varphi$  is  $p$ -vacuous in  $BMC_k(K, \varphi)$ , it means that  $\varphi$  is  $k$ -vacuous, and  $p$  is

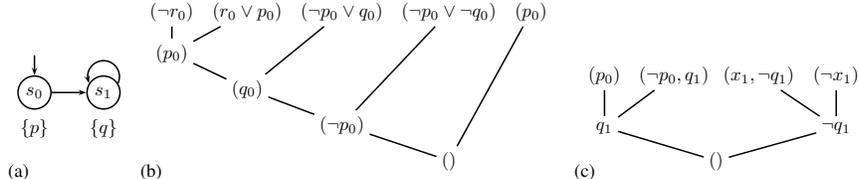


Fig. 1. (a) A Kripke structure; (b) A resolution proof for EXAMPLE 2; (c) A resolution proof for EXAMPLE 1.

such that  $K \models_k \varphi[p \leftarrow x]$ , where  $x$  is a new unconstrained variable of  $K$ .

Def. 1 suggests a sound and complete algorithm for vacuity detection: for each propositional variable  $p$  in  $\varphi$ , run BMC on  $\varphi[p \leftarrow x]$ , where  $x$  is a variable that does not appear in  $K$  and  $\varphi$ . If  $K \models_k \varphi[p \leftarrow x]$  for some  $p$ , then  $\varphi$  is  $k$ -step vacuous. We refer to this algorithm as *naive*. Its drawback is that it may require as many model-checking runs as there are propositional variables in  $\varphi$ . Defs. 1 and 2 can be generalized to vacuity in arbitrary (not necessarily atomic) subformulas. This follows from the fact that a subformula is vacuous iff it is *mutually vacuous* in all of its atomic propositions [14, Th. 9], and that the definitions can be easily extended to mutual vacuity. For example, if  $\varphi$  contains subformula  $\theta = p \wedge q$ , and  $p$  and  $q$  are mutually vacuous, then we can deduce that  $\theta$  is vacuous as well.

We now review some of the alternative definitions of vacuity and their algorithms. The first attempt to formulate and automate vacuity detection is due to Beer et al. [2]. They consider a property  $\varphi$  to be vacuous if  $\varphi$  contains a subformula  $\psi$  such that replacing  $\psi$  by any other formula does not affect the satisfaction of  $\varphi$ . Applying this definition directly would require an infinite number of subformula replacements, precluding a practical implementation. However, Beer et al. show that to detect vacuity w.r.t. a *single* occurrence of a subformula  $\psi$  in w-ACTL, it is sufficient to replace  $\psi$  with only true and false. This was later extended to CTL\* by Kupferman and Vardi [3]. Purandare and Somenzi [4] showed how to speed up subformula vacuity by analyzing the parse tree of a CTL property.

Armoni et al. [5] generalized the above syntactic definition of vacuity by introducing universal quantification, i.e.,  $\forall x \cdot \varphi[\psi \leftarrow x]$ . Based on the domain of  $x$ , three notions of vacuity are obtained, the most robust of which being *trace* vacuity. Gurfinkel and Chechik [15] extended Armoni's definition of vacuity to CTL\*, thus uniformly capturing CTL and LTL. Armoni et al. also analyzed the syntactic structure of the property in order to avoid checking the operands of subformulas that are known to be vacuous. Such optimizations complement our techniques, which focus on detecting vacuous atomic subformulas.

Namjoshi [16] defines a somewhat different notion of vacuity, also based on a proof derived from a successful model checking run. According to Namjoshi, a property should only be considered vacuous if every proof of why it holds on the model exhibits vacuity. This definition of vacuity coincides with the definition of [5], [15] for a subset of LTL. Our methods efficiently examine proofs derived from model-checking runs, but are able to detect vacuity as defined by [2], [5], [15], [17]. Finally, we cannot empirically compare our techniques, since

no experimental results are provided in [16].

Our definition of vacuity is syntactic, and in this respect, it is similar to the original definition of Beer et al. [2]. However, Def. 1 is stronger, and is equivalent to the semantic definition of Armoni et al. [5], as shown by Gurfinkel and Chechik [15].

#### IV. EXPLOITING RESOLUTION PROOFS

In Sec. III, we discussed the existence of a sound and complete vacuity detection algorithm for BMC, which requires as many model-checking runs as there are propositional variables in the property being checked. We propose a new vacuity detection strategy: first detect partial vacuity using inexpensive techniques and then complete the analysis using extra model-checking runs. Since we are interested in replacing expensive model-checking runs by inexpensive partial vacuity detection methods, we limit ourselves to considering the output of the original model-checking run on  $BMC_k(K, \varphi)$ , i.e.,  $CL_K \cup CL_e$ . This run provides us with a single resolution proof to analyze, but in general, there may be many ways to derive the empty clause from different subsets of  $BMC_k(K, \varphi)$ . Any method that only examines one of these derivations is inherently incomplete, in the sense that a property may be  $p$ -vacuous but there is no way of determining this based on a given resolution proof. For example, consider a model that is composed of two completely disjoint sub-models, running in parallel, i.e.,  $K = K_1 \parallel K_2$ . Suppose that  $K_1$  satisfies  $\mathbf{G}p$ ,  $K_2$  satisfies  $\mathbf{G}q$ , and that both do so non-vacuously. Then the property  $\varphi = \mathbf{G}p \vee \mathbf{G}q$  holds on  $K$   $p$ -vacuously and  $q$ -vacuously. However, one of the possible resolution proofs showing that  $\varphi$  holds proves that  $\mathbf{G}p$  holds non-vacuously on  $K_1$ . Thus, it is impossible to determine that  $\varphi$  is vacuous in  $p$  from this proof. Any method based on examining only one resolution proof cannot prove the absence of vacuity, since another resolution proof, showing the property to be vacuous, might exist.

In this section, we introduce three algorithms of increasing precision for partial vacuity detection, based on examining the UNSAT core (irrelevance and local irrelevance) and the resolution proof produced by BMC (peripherality).

##### A. Examining UNSAT cores

Given a resolution proof that  $BMC_k(K, \varphi)$  is unsatisfiable, we can sometimes cheaply determine that the similar theory  $BMC_k(K, \varphi[p \leftarrow x])$  is also unsatisfiable, and therefore, that the property is  $p$ -vacuous. In this section, we consider how to determine that  $BMC_k(K, \varphi[p \leftarrow x])$  is unsatisfiable given that  $BMC_k(K, \varphi)$  is unsatisfiable, using only an UNSAT core.

1) *Irrelevance*: Intuitively, any variable that does not appear in the UNSAT core does not contribute to the reason why  $\varphi$  holds on  $K$ , so it can be considered *irrelevant*.

**Definition 3** Let  $K$  be a model, and  $\varphi$  an LTL formula. Assume that  $\Pi$  is an UNSAT core of  $BMC_k(K, \varphi)$  witnessing that  $K \models_k \varphi$ . Then,  $p$  is irrelevant with respect to  $BMC_k(K, \varphi)$  and  $\Pi$  iff  $p_i$  does not appear in  $\Pi$  for any time instance  $i$ .

If a variable is irrelevant, it is also vacuous, as shown by the following theorem. The proofs of this and other theorems are given in Appendix A.

**Theorem 1** If  $p$  is irrelevant with respect to  $BMC_k(K, \varphi)$  and  $\Pi$ , then  $\varphi$  is  $k$ -step  $p$ -vacuous.

Def. 3 provides an algorithm to detect some vacuous variables. However, a variable can appear in the UNSAT core and still be vacuous, as demonstrated by the following example.

**EXAMPLE 1.** Consider a Kripke structure  $K$  with variables  $p$  and  $q$  given by the constraints  $Init = p \wedge q$ ,  $R = p \Rightarrow q'$ , which mean that the initial state is labeled by  $\{p, q\}$ , and the transition relation is expressed by the propositional formula  $p \Rightarrow q'$  over unprimed and primed variables. Let  $\varphi = X(p \vee q)$  be the property to check.  $\varphi$  is  $p$ -vacuous since it is satisfied simply because  $q$  is true in any successor of the initial state. The CNF encoding of the one-step BMC problem is  $CL_K = \{(p_0 \wedge q_0), (p_0 \Rightarrow q_1)\} = \{(p_0), (q_0), (\neg p_0, q_1)\}$ ,  $CL_e = \{(\neg p_1), (p_1, \neg q_1)\}$ . In this case, the *unique* minimal UNSAT core contains all of the clauses of the problem except for  $(q_0)$ . Thus, all  $p_i$  appear in the UNSAT core, and  $p$  cannot be determined vacuous using irrelevance.  $\square$

This example shows that even if we are to look at every UNSAT core of a BMC problem, irrelevance is still unable to detect existing vacuity.

2) *Local Irrelevance:* Variables which do not appear in the UNSAT core are vacuous. The converse is not true: vacuous variables may also appear in the UNSAT core. Intuitively, these variables are not the central reason why  $\varphi$  holds on  $K$ . For example, the clauses of  $CL_K$  may resolve against each other, representing some simplification and unification of parts of the model, before resolutions with  $CL_e$  clauses are performed. If a variable is resolved upon using only the  $CL_K$  clauses or only the  $CL_e$  clauses, it is potentially vacuous. By looking at the UNSAT core, it is possible to anticipate whether a variable will not be involved in resolutions between  $CL_K$  and  $CL_e$  using the following definition.

**Definition 4** Let  $K$  be a model, and  $\varphi$  an LTL formula. Assume that  $\Pi$  is an UNSAT core of  $BMC_k(K, \varphi)$  witnessing  $K \models_k \varphi$ . Then,  $p$  is locally irrelevant with respect to  $BMC_k(K, \varphi)$  and  $\Pi$  iff for each time instance  $i$ , either  $p_i$  does not appear in  $\Pi$  or  $p_i$  is local to either  $CL_e \cap \Pi$  or  $CL_K \cap \Pi$ .

In Example 1,  $p$  is locally irrelevant since  $p_1$  only occurs in the clauses of  $U$  taken from  $CL_e$ , while  $p_0$  only appears in  $U$  within  $CL_K$  clauses. Moreover, the UNSAT core of the original problem can be converted to an UNSAT core of the new theory, thus proving that  $p$  is vacuous. Specifically,  $U = \{(p_0), (\neg p_0, q_1), (\neg p_1), (p_1, \neg q_1)\}$  is the UNSAT core of the original problem, so substituting  $x$  for  $p$  in the clauses of  $U$  that came from  $CL_e$  gives  $U' = \{(p_0), (\neg p_0, q_1), (\neg x_1), (x_1, \neg q_1)\}$ . This is a subset of

$BMC_1(K, \varphi[p \leftarrow x]) = \{(p_0), (q_0), (\neg p_0, q_1), (\neg x_1), (x_1, \neg q_1)\}$ , so it is a candidate for the new UNSAT core. The substitution may have prevented the resolutions necessary to derive the empty clause. However, Fig. 1(c) shows a proof that  $U'$  is also unsatisfiable. In this case, it was possible to substitute  $x_i$  for  $p_i$  in the clauses coming from  $CL_e$  in the original UNSAT core and create an UNSAT core for  $BMC_k(K, \varphi[p \leftarrow x])$ . In fact, this observation applies to all cases of local irrelevance by Theorem 2. Therefore, Def. 4 specifies an algorithm to detect some vacuous variables.

**Theorem 2** If  $p$  is locally irrelevant with respect to  $BMC_k(K, \varphi)$  and  $\Pi$ , then  $\varphi$  is  $k$ -step  $p$ -vacuous.

Unfortunately, if a variable  $p$  is not locally irrelevant in an UNSAT core, the formula can still be  $p$ -vacuous, as shown by the following example.

**EXAMPLE 2.** Consider a Kripke structure with atomic propositions  $r$ ,  $p$  and  $q$  whose initial state is given by the constraint:  $Init = \neg r \wedge p \wedge q$ . The formula  $\varphi = \neg p \vee q$  is  $p$ -vacuous in the initial state. Let us assume that the zero-step BMC problem is encoded in CNF as follows:

$$CL_K = (\neg r_0)(r_0 \vee p_0)(\neg p_0 \vee q_0)$$

$$CL_e = (p_0)(\neg p_0 \vee \neg q_0)$$

There are several resolution proofs that can establish unsatisfiability of  $CL_K \cup CL_e$ ; one such proof is shown in Fig. 1(b). In none of the proofs is  $p$  locally irrelevant with respect to  $CL_e$  and  $CL_K$ .  $\square$

The problem with local irrelevance is that it is impossible to tell if a variable is going to be used in a resolution joining  $CL_K$  and  $CL_e$  clauses based on the UNSAT core alone.

### B. Peripherality

In Sec. IV-A, two vacuity detection methods based on examining the variables in the UNSAT core were found to fall short of completeness. It was seen that even if every possible resolution proof could be analyzed, irrelevance and local irrelevance still might fail to detect existing vacuity. Here, we extend the analysis to the resolution proof's structure. The resulting peripherality algorithm is superior, since it guarantees vacuity will be found if all possible resolution proofs are considered.

The limitations of detecting vacuity based only on the UNSAT core were demonstrated in Example 2. By examining the resolution proof in Fig. 1(b), we see that although  $p_0$  appears both in  $CL_K$  clauses and in  $CL_e$  clauses, it is always resolved "locally". That is, if we resolve two clauses  $c_1 = (\dots, p_i, \dots)$  and  $c_2 = (\dots, \neg p_i, \dots)$ ,  $p_i$  and  $\neg p_i$  must have been preserved from their original source in some set of root clauses. If all the originating root clauses belong to  $CL_K$  or all belong to  $CL_e$ , then  $p_i$  is being resolved on locally. In this case, we can replace  $p_i$  in either set of clauses without affecting their unsatisfiability. For example, in Fig. 1(b),  $p_0$  can be replaced in  $CL_e$  by a new unconstrained variable  $x_0$ . This intuition is formalized below.

Given a resolution proof  $\Pi$ , a variable  $l$ , and a clause  $c$ , we denote by  $S(l, c)$  the set of all root clauses that have contributed the variable  $l$  to  $c$ .  $S(l, c)$  is defined recursively as shown in Fig. 3. A root clause  $c_r$  is an element of  $S(l, c)$  if it contains

$$\begin{array}{l}
L(c) : \text{clause } c, \text{ variable } p \rightarrow \{\emptyset, \text{'A'}, \text{'B'}, \text{'AB'}\} \\
\bullet \text{ if } c \in \text{Roots}(\Pi) \text{ then} \\
L(c) = \begin{cases} \emptyset & \text{if } p \notin c \\ \text{'A'} & \text{if } p \in c \wedge c \in A \\ \text{'B'} & \text{if } p \in c \wedge c \in B \end{cases} \\
\bullet \text{ else if } c \text{ is a clause resulting from resolving } c_1 \text{ and } c_2 \text{ on variable } v, \text{ i.e., } c = \exists v \cdot c_1 \wedge c_2, \text{ then} \\
\text{ - if } v \neq p, \text{ then} \\
L(c) = \begin{cases} \emptyset & \text{if } L(c_1) = L(c_2) = \emptyset \\ \text{'A'} & \text{if } \exists i, j \cdot L(c_i) = \text{'A'} \wedge L(c_j) \subseteq \{\text{'A'}, \emptyset\} \\ \text{'B'} & \text{if } \exists i, j \cdot L(c_i) = \text{'B'} \wedge L(c_j) \subseteq \{\text{'B'}, \emptyset\} \\ \text{'AB'} & \text{otherwise} \end{cases} \\
\text{ - else if } v = p, \text{ then} \\
L(c) = \begin{cases} \emptyset & \text{if } L(c_1) = L(c_2) \\ \text{'AB'} & \text{otherwise} \end{cases}
\end{array}$$

Fig. 2. Labeling function for the peripherality algorithm.

$$S(l, c) = \begin{cases} \emptyset & \text{if } l \notin c \\ c & \text{if } c \in \text{Roots}(\Pi) \wedge l \in c \\ S(l, c_1) \cup S(l, c_2) & \text{if } c_1 \text{ and } c_2 \text{ are parents} \\ & \text{of } c \wedge l \in c \end{cases}$$

Fig. 3. Definition of  $S(l, c)$ .

a variable  $l$  and there exists a path from  $c_r$  to  $c$  that does not contain a resolution on  $l$ . We can now define *peripherality* of variables, which captures the conditions when a global variable may not be central to the reason why  $\varphi$  holds on  $K$ .

**Definition 5** Let  $A$  and  $B$  be disjoint sets of clauses such that  $C = A \cup B$  is unsatisfiable, and  $\Pi$  be a resolution proof establishing unsatisfiability of  $C$ . Then a variable  $l$  is *peripheral* with respect to  $A$  and  $B$  iff for every resolution on  $l$  between clauses  $c_1$  and  $c_2$ ,  $S(l, c_1) \cup S(l, c_2) \subseteq A$  or  $S(l, c_1) \cup S(l, c_2) \subseteq B$ .

Within the BMC setting, we have the following definition:

**Definition 6** Let  $K$  be a model,  $\varphi$  be an LTL formula,  $\text{BMC}_k(K, \varphi)$  be a CNF encoding of a BMC problem for  $K \models_k \varphi$ , and  $\Pi$  be a proof of unsatisfiability of  $\text{BMC}_k(K, \varphi)$ .  $p$  is peripheral in  $\varphi$  iff for each time instance  $i$ ,  $p_i$  is peripheral in  $\Pi$  with respect to  $CL_e$  and  $CL_K$ .

If a variable is peripheral, it is vacuous by Theorem 3.

**Theorem 3** Let  $\Pi$  be a proof of unsatisfiability of  $\text{BMC}_k(K, \varphi)$ . If a variable  $p$  of  $\varphi$  is peripheral in  $\Pi$ , then  $\varphi$  is  $k$ -step  $p$ -vacuous.

In Fig. 1(b), although  $p$  is not locally irrelevant in  $\varphi$ , it is peripheral, and therefore  $\varphi$  is  $p$ -vacuous. This also demonstrates that peripherality is a strictly stronger notion than local irrelevance. Theorem 4 shows that under our constraints this is the strongest result that we can hope to establish.

**Theorem 4** Assume  $\varphi$  is  $k$ -step  $p$ -vacuous in  $K$ . Then, there exists a resolution proof  $\Pi$  of unsatisfiability of  $\text{BMC}_k(K, \varphi)$  such that  $p$  is peripheral in  $\Pi$ .

This is one of the main contributions of this paper: if a variable appears in all proofs, but is detected as peripheral in at least one of these proofs, it is vacuous. Conversely, if a variable appears in all proofs but is not peripheral in any of them, it is definitively not vacuous.

Peripherality of a variable can be detected by traversing the

resolution proof from the roots to the leaf, keeping track of the source of the variable in each clause. If  $\Pi$  is a resolution proof whose root clauses are divided into two disjoint sets,  $A \cup B$ , then the labeling function  $L$  is defined recursively as shown in Fig. 2, where  $c$  is used to represent a clause. This labeling function defines an algorithm for detecting peripherality.

A CNF variable  $v$  is peripheral iff the label of the empty clause is not ‘AB’. Thus, to detect whether a formula  $\varphi$  is  $p$ -vacuous, we need to check that all CNF variables  $p_i$  corresponding to  $p$  (see Sec. II) are peripheral. This can be done by applying the labeling function described in Fig. 2 with  $A = CL_K$ , and  $B = CL_e$  for each  $p_i$  (for details, see [8]). It is also possible to simultaneously keep track of the labels for all CNF variables so that only a single pass through  $\Pi$  is needed. The time complexity of the peripherality algorithm is linear in the size of the resolution proof.

**Theorem 5** For a resolution proof  $\Pi$  that  $\text{BMC}_k(K, \varphi)$  is unsatisfiable, determining which variables of  $\varphi$  are peripheral can be done in time linear in the size of  $\Pi$ .

In this section, we defined three methods of detecting vacuity based on examining the UNSAT core and the resolution proof produced by BMC. Our evaluation of these algorithms w.r.t. precision and execution times can be found in Sec. V.

## V. PRACTICAL EXPERIENCE

The techniques reported in this paper have been implemented in a tool called **VaqTree** (see [8] for a description of this tool). The inputs to **VaqTree** are a model (encoded using the language of NuSMV [10]) and an LTL property. The tool returns the vacuity status of each variable in the property. Vacuity detection in **VaqTree** proceeds in two phases: a ‘‘partial pass’’ that applies one of our methods, and a ‘‘model-checking pass’’ that completes the analysis using additional model-checking runs.

We have run **VaqTree** on two benchmark suites. To evaluate the overall performance of the tool and the effectiveness of our partial vacuity detection methods, we have created a benchmark suite  $\mathcal{S}_A$  using various models and properties from the NUSMV distribution. To evaluate the scalability of the tool to industrial models, we have created a benchmark suite  $\mathcal{S}_B$  from the models in the IBM Formal Verification Benchmarks Library [18].

These models came with rather simple properties (one temporal operator), and (as expected from an industrial benchmark) did not exhibit a high degree of vacuity. Thus, we used this suite to measure the “worst-case” behavior of the tool, i.e., the amount of overhead incurred by our methods when no vacuity is found.

In the benchmarks, each test case consists of a model  $M$ , a property  $\varphi$ , and a bound  $k$  such that  $M \models_k \varphi$ . Note that finding an appropriate bound  $k$  is orthogonal to  $k$ -vacuity detection, which explains why our evaluation does not consider the time needed to find  $k$ . The experiments were performed on a Linux machine with a 2.8GHz P4 CPU, and 1GB of RAM, with up to 700MB of RAM available to each process. Currently, *VaqTree* is limited to proofs with up to 2.5 million resolutions. In  $\mathcal{S}_A$ , this corresponds to a test case from the asynchronous **abp4** model (roughly 30 boolean variables, with  $k = 19$ ). A sample of our experimental data is available in Appendix B, and the full results – in [19]. Below, we discuss results obtained with each benchmark individually.

#### A. Results obtained with $\mathcal{S}_A$

This benchmark suite consists of 5 models: **abp4**, **msi\_wtrans**, **pai**, and **prod-cell** from the NUSMV distribution (107 properties) and **toyFGS04** from [20] (14 properties). On average, the properties in the suite have 2 temporal operators (from the set G, F, U and X), with a maximum of 4 operators, and include both liveness and safety. 99 of the properties exhibit vacuity, and 22 do not.

Scatter plots in Fig. 4 compare the execution times of *VaqTree* (parametrized with irrelevance, local irrelevance, and peripherality), with naive detection for this benchmark. Execution times for naive detection include CNF theory generation and satisfiability testing for each variable of the property. Execution times for *VaqTree* include the time for the partial pass and the subsequent model-checking pass. Each point in the plot represents a single test case. The X-axis represents the time (in seconds) taken by naive detection. The Y-axis represents the time (in seconds) taken by *VaqTree* when parameterized by each of our methods. Points below the diagonal indicate where *VaqTree* was faster than naive detection; points near the diagonal indicate cases where the partial pass found a small percentage of the vacuous variables.

Fig. 5 shows that on  $\mathcal{S}_A$ , *VaqTree* with irrelevance finds the fewest vacuous variables among our partial methods, as expected from the discussion in Section IV. Although Fig. 4(b) and (c) look similar, the numbers (see Appendix B and [19]) show that local irrelevance is faster than peripherality in 96% of the cases. This is consistent with the additional work peripherality must perform to analyze the proof tree. A detailed comparison of local irrelevance and naive detection shows that *VaqTree* with local irrelevance was faster or comparable to naive detection in 95% of the test cases. *VaqTree* with local irrelevance was faster than naive detection in 70 (58%) of the test cases, out of which 30 cases were twice as fast, and 20 cases were faster by an order of magnitude. In the remaining 51 cases, local irrelevance was at most 3% slower in 86% of these cases.

There are 10 cases where *VaqTree* with peripherality took much longer than naive detection. All of these cases are from

the **abp4** model, and while they have the largest resolution proofs of the benchmark suite (between 300,000 and 2M clauses), other 300,000-clause test cases did not yield poor performance. We conjecture that the poor performance is due to a low clause/variable ratio [21] which favours naive detection in cases where vacuity is not present. Intuitively, a low ratio indicates that the SAT instance is underconstrained, and so a solution (if it exists) can be found quickly. On the other hand, finding a proof of *unsatisfiability* in a model with few constraints can be more difficult. Naive detection on a non-vacuous property requires solving satisfiable SAT instances, since replacing variables falsifies the property. However, peripherality on a non-vacuous property requires time linear in the size of the resolution proof obtained from the original model-checking run. If all of these SAT instances have a low clause/variable ratio, naive detection can be much faster than peripherality. This situation was only observed on the **abp4** model, with clause/variable ratio of 1.5-1.8 – significantly lower than any other test case with large proofs and without vacuity.

We now turn to measuring the effectiveness of our methods, using the number of vacuous variables found during the partial pass as a metric (see the scatter plots in Fig. 5). This number indicates how many additional model-checking runs are needed to complete vacuity detection. Since our partial methods can be ordered by increasing precision, Fig. 5(a) compares irrelevance and local irrelevance, Fig. 5(b) – local irrelevance and peripherality, and Fig. 5(c) – peripherality and naive detection. Each point in the plot represents a set of test cases – a larger point means a larger set. The axes show the number of vacuous variables detected by each method. Points below the diagonal indicate where the X-axis method detects more vacuous variables than the Y-axis method. The plots show that local irrelevance is clearly more effective than irrelevance. Contrary to our expectations, peripherality performed exactly as local irrelevance in all but 5 cases. Thus, local irrelevance appears to be more cost-effective. Fig. 5(c) shows that our techniques are effective when compared with naive detection: peripherality reduced the number of extra model-checking runs by 40% in 54 out of 99 cases that exhibited vacuity.

#### B. Results obtained with $\mathcal{S}_B$

This benchmark suite consists of 13 models from the IBM Formal Verification Benchmarks Library [22] (18 properties). The properties have a single temporal operator (G or F), and include both safety and liveness. 12 of the properties exhibit vacuity, and 6 do not. To evaluate the scalability of *VaqTree* to industrial models, we must first bound such that  $M \models_k \varphi$ . For this benchmark, we picked depth  $k = 20$ , which is in line with the bounds used for analyzing these models in [22, Sec. 2]. At this depth, only 13 models from the benchmark were suitable for our experiments. We report on the experiments below. At this  $k$ , some of the models were too large to analyze using *VaqTree*, and some of the properties did not hold. This is why we only report data for 13 models from this benchmark.

Table I, which includes full results for  $\mathcal{S}_B$ , shows that proof sizes for this benchmark can be handled by *VaqTree*. Interest-

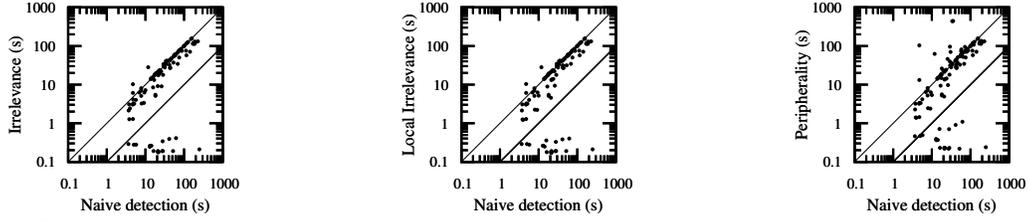


Fig. 4.  $\mathcal{S}_A$ : Comparison of execution times. Where applicable, all times include times for both the partial and model-checking passes.

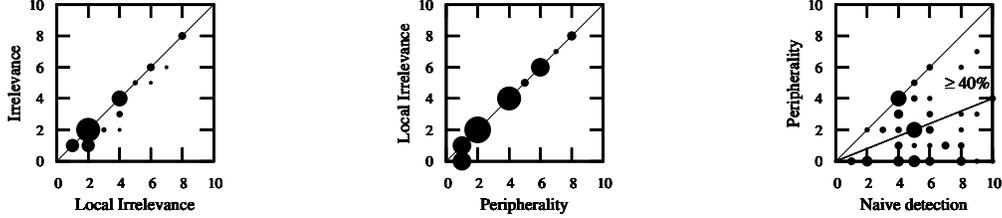


Fig. 5.  $\mathcal{S}_A$ : Comparison of the number of vacuous variables detected by partial pass. Larger points represent more test cases than the smaller points.

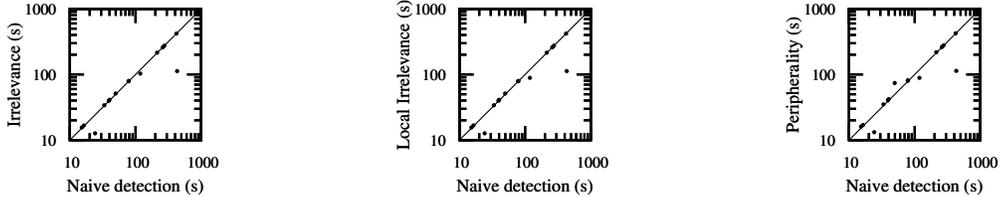


Fig. 6.  $\mathcal{S}_B$ : Comparison of execution times. Where applicable, all times include times for both the partial and model-checking passes.

ingly, these are in the same range as proof sizes for  $\mathcal{S}_A$ . This could be explained by the fact that even though these models are more complex, properties are simpler.

Scatter plots in Fig. 6 compare the execution times of *VaqTree* parametrized with local irrelevance and peripherality, with naive detection for this benchmark. Execution times are measured as described in Section V-A, and the graphs are interpreted in the same way as those in Fig. 4. Since  $\mathcal{S}_B$  had low vacuity, we did not expect our techniques to find it without the help of naive detection. However, graphs in Fig. 6 show that our techniques do in fact detect vacuity, as indicated by the points that appear below the diagonal. Both local irrelevance and peripherality detect the same amount of vacuity in  $\mathcal{S}_B$ , but local irrelevance is slightly faster than peripherality.

Surprisingly, peripherality introduces a low overhead in this benchmark – points over the diagonal are near it, unlike what we see in Fig. 4. To explain this behavior, we hypothesized that in non-vacuous cases with low clause/variable ratios and large proofs, peripherality is much slower than naive detection. In  $\mathcal{S}_B$ , we found that 15 of the test cases have a clause/variable ratio between 2.62-3.66, much higher than the ratios encountered in  $\mathcal{S}_A$ . The remaining three cases had ratios in the same range as the **abp04** model. However, two of these produce trivial proofs, and the last one exhibits vacuity. These results empirically support our hypothesis.

### C. Conclusions

In summary, we observed that local irrelevance performs best out of our proposed partial methods, finding most vacuity in

the least amount of time. In 95% of both benchmark suites, we found *VaqTree* with local irrelevance to be at most 3% slower, and usually much faster, than the naive detection. In several tests of the  $\mathcal{S}_A$  benchmark, peripherality was noticeably slower than naive detection. On the industrial benchmark  $\mathcal{S}_B$ , the overhead produced by peripherality was negligible. Interestingly, this suggests that peripherality may be a viable alternative to local irrelevance on industrial models. We plan to investigate this further in the future. Thus, we believe that both local irrelevance and peripherality can be used to replace naive detection. We plan to enhance our methods by developing a heuristic based on the clause/variable ratio and proof size that indicates when naive detection should be applied instead. Finally, *VaqTree* outputs the vacuity results for each timed variable  $p_i$  as a byproduct of its partial pass. This information gives an *explanation of non-vacuity*, indicating which time steps have been important for deciding whether a given variable was vacuous, thus facilitating debugging.

## VI. SUMMARY AND RELATED WORK

In this paper, we showed how to exploit the UNSAT core and resolution proof produced by a successful run of BMC for vacuity detection. We introduced three vacuity detection methods that can be applied with little overhead after one model-checking run in order to quickly identify vacuous variables and reduce the number of additional model-checking runs required. Two of these methods, irrelevance and local irrelevance, exploit the UNSAT core, and the third, peripherality, is based on analyzing the resolution proof. We built a tool *VaqTree*, which

is based on these methods, and showed that it is effective for speeding up vacuity detection.

Related work on vacuity detection has been described in Section III. Additionally, our work is related to research in declarative modeling. In particular, our use of the UNSAT core to detect vacuity was inspired by [23], which addresses the problem of identifying overconstraint in declarative models. While similar in spirit to vacuity detection in model checking, declarative models have no explicit transition relation; instead, transitions are expressed with constraints [24], [25]. An overconstraint occurs when the model satisfies a safety property because all violations of the formula have been accidentally ruled out by the declared constraints. In order to detect such overconstraints, [23] introduces the idea of *core extraction*: declarative models are reduced to SAT instances, from which an UNSAT core can be extracted if the property holds. If a constraint's clauses do not appear in the UNSAT core, the constraint is called *irrelevant*, and is a source of overconstraint (similar to Def. 3). The cone-of-influence technique [1] is also similar to Def. 3. However, as both of these techniques are model-based, neither can be used to detect vacuity.

Our experiments show that local irrelevance and peripherality can detect more vacuous variables than irrelevance. Therefore, detecting overconstraint in declarative models may also benefit from methods that analyze the structure of the resolution proof. In the future, we propose to investigate how a notion equivalent to peripherality can be defined in the declarative setting. Another goal of future work is to increase the power of resolution proof-based vacuity detection methods. In this paper, we restricted ourselves to using results of only one BMC run, and to methods with linear time complexity in the size of the resolution proof or better. However, it is possible that the most optimal trade-off between speed and effectiveness of vacuity detection algorithms lies in the domain of multiple resolution proofs, where we can find the minimal UNSAT core [26] or reduce the resolution proof using interpolation [27].

McMillan [6] uses interpolation to prove that a particular bound is sufficient to imply the unbounded satisfaction of a BMC problem. We intend to combine our techniques with this algorithm in order to prove that bounded vacuity for the correct  $k$  implies that the property also holds vacuously in the unbounded case.

Interpolation can also be used to detect vacuity. Given two sets of clauses,  $A$  and  $B$ , such that  $A \cup B$  is unsatisfiable, an interpolant  $C$  is a set of clauses whose variables appear in both  $A$  and  $B$ , such that  $B \cup C$  is unsatisfiable and  $A \Rightarrow C$  [28]. Intuitively, if  $C$  is minimal, then  $C$  is the reason why  $A \cup B$  is unsatisfiable. This intuition suggests that if an interpolant of  $CL_K$  and  $CL_e$  could be found, then all variables not appearing in it could be considered vacuous. However, we did not include this technique in our empirical evaluation, as our interpolant generator was comparatively slower.

Another means of speeding vacuity detection for BMC is to iteratively check the  $k$ -step vacuity of each variable starting with  $k = 0$ . Since  $K \not\models_{k_1} \varphi[p \leftarrow x]$  implies  $K \not\models_{k_2} \varphi[p \leftarrow x]$  for all  $k_2 > k_1$ , if a variable is proven non-vacuous at some

step  $k$ , then it can be omitted from subsequent checks of higher  $k$ . This method is orthogonal to our techniques, and the vacuity detection at each step could be carried out by *VaqTree*.

## REFERENCES

- [1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
- [2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient Detection of Vacuity in ACTL Formulas," in *Proc. of CAV'97*, ser. LNCS, vol. 1254, 1997, pp. 279–290.
- [3] O. Kupferman and M. Vardi, "Vacuity Detection in Temporal Model Checking," in *Proc. of CHARME'99*, ser. LNCS, vol. 1703, 1999.
- [4] M. Purandare and F. Somenzi, "Vacuum Cleaning CTL Formulas," in *Proc. of CAV'02*, ser. LNCS, vol. 2404, 2002, pp. 485–499.
- [5] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi, "Enhanced Vacuity Detection in Linear Temporal Logic," in *Proc. of CAV'03*, vol. 2725, July 2003, pp. 368–380.
- [6] K. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. of CAV'03*, ser. LNCS, vol. 2725, July 2003, pp. 1–13.
- [7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proc. of TACAS'99*, ser. LNCS, vol. 1579, 1999.
- [8] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik, "Exploiting Resolution Proofs for LTL Vacuity Detection," [ftp://ftp.cs.toronto.edu/pub/reports/csr/547/TR-547.ps](http://ftp.cs.toronto.edu/pub/reports/csr/547/TR-547.ps), DCS, University of Toronto, CSR TR 547, 2007.
- [9] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani, "Improving the Encoding of LTL Model Checking into SAT," in *Proc. of VMCAI'02*, ser. LNCS, vol. 2294, 2002, pp. 196–207.
- [10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. of CAV'02*, ser. LNCS, vol. 2404, 2002, pp. 359–364.
- [11] N. Een and N. Sörensson, "The MiniSat Page," <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html>, April 2006.
- [12] L. Zhang and Z. Fu, "Boolean Satisfiability Research Group at Princeton," <http://www.princeton.edu/~chaff/>, September 2006.
- [13] L. Zhang and S. Malik, "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications," in *Proc. of DATE'03*, 2003, pp. 10 880–10 885.
- [14] A. Gurfinkel and M. Chechik, "How Vacuous Is Vacuous?" in *Proc. of TACAS'04*, ser. LNCS, vol. 2988, March 2004, pp. 451–466.
- [15] —, "Extending Extended Vacuity," in *Proc. of FMCAD'04*, ser. LNCS, vol. 3312, 2004, pp. 306–321.
- [16] K. Namjoshi, "An Efficiently Checkable, Proof-Based Formulation of Vacuity in Model Checking," in *Proc. of CAV'04*, ser. LNCS, vol. 3114, 2004, pp. 57–69.
- [17] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient Detection of Vacuity in Temporal Model Checking," *FMSD*, vol. 18, no. 2, 2001.
- [18] I. Haifa, "CNF Benchmarks from IBM Formal Verification Benchmarks Library," 2007. [Online]. Available: [http://www.haifa.il.ibm.com/projects/verification/RB\\_Homepage/benchmarks.html](http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html)
- [19] J. Simmonds and J. Davies, "VaqTree," <http://www.cs.toronto.edu/~jsimmond/VaqTree>, 2007.
- [20] M. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-generating Test Sequences Using Model Checkers: A Case Study," in *Proc. of FATES'03*, ser. LNCS, vol. 2931, 2003, pp. 42–59.
- [21] B. Selman, D. Mitchell, and H. Levesque, "Generating Hard Satisfiability Problems," *Artificial Intelligence*, vol. 81, no. 1-2, pp. 17–29, 1996.
- [22] E. Zarpas, "Benchmarking SAT Solvers for Bounded Model Checking," in *SAT*, 2005, pp. 340–354.
- [23] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri, "Debugging Overconstrained Declarative Models Using Unsatisfiable Cores," in *Proc. of ASE'03*, October 2003, pp. 94–105.
- [24] D. Jackson, "Alloy: a Lightweight Object Modelling Notation," *ACM TOSEM*, vol. 11, no. 2, pp. 256–290, 2002.
- [25] J. M. Spivey, *The Z Notation: a Reference Manual*. Prentice Hall, 1992.
- [26] R. Gershman, M. Koifman, and O. Strichman, "Deriving Small Unsatisfiable Cores with Dominators," in *Proc. of CAV'06*, ser. LNCS, vol. 4144, 2006, pp. 109–122.
- [27] W. Craig, "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem," *JSL*, vol. 22, pp. 250–268, 1957.
- [28] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan, "Abstractions from Proofs," in *Proc. of POPL'04*. ACM, January 2004, pp. 232–244.

## A. Proofs of Theorems

Proofs of selected theorems are given. Additional proofs can be found in [8, Appendix A2].

**Theorem 2** *If  $p$  is locally irrelevant with respect to  $BMC_k(K, \varphi)$  and  $\Pi$ , then  $\varphi$  is  $k$ -step  $p$ -vacuous.*

*Proof:* Let  $BMC_k(K, \varphi) = CL_K \cup CL_e$  and  $U$  be the UNSAT core of  $\Pi$ . Assume that  $p$  is locally irrelevant in  $BMC_k(K, \varphi)$ . So for all  $p_i$ , either  $p_i$  does not appear in  $U$ , or  $p_i$  is local to  $CL_e \cap U = U_e$  or to  $CL_K \cap U = U_K$  by Def. 4. Let  $U_{e'}$  be  $U_e$  with each occurrence of  $p_i$  replaced by  $x_i$ . Since each  $p_i$  that has been replaced is local to  $U_e$ , and  $U_K \cup U_e = U$  is unsatisfiable, then  $U_K \cup U_{e'}$  is also unsatisfiable. Since  $U_{e'} \subseteq CL_e[p \leftarrow x]$ , the set of clauses  $CL_K \cup CL_e[p \leftarrow x]$  is unsatisfiable as well. Therefore,  $K \models_k \varphi[p \leftarrow x]$  holds, so  $\varphi$  is  $p$ -vacuous.  $\square$

**Theorem 3** *Let  $\Pi$  be a proof of unsatisfiability of  $BMC_k(K, \varphi)$ . If a variable  $p$  of  $\varphi$  is peripheral in  $\Pi$ , then  $\varphi$  is  $k$ -step  $p$ -vacuous.*

*Proof:* Let  $BMC_k(K, \varphi) = CL_K \cup CL_e$  and  $U$  be the UNSAT core of  $\Pi$ . Assume that  $p$  is peripheral in  $BMC_k(K, \varphi)$ . Let  $U_{e'}$  be the result of replacing each  $p_i$  with  $x_i$  in  $CL_e \cap U$ . Then  $(CL_K \cap U) \cup U_{e'}$  is still unsatisfiable, since every resolution on  $x_i$  must be local to  $CL_e \cap U$ , and every resolution on  $p_i$  must be local to  $CL_K \cap U$  by the peripherality of  $p_i$ . Since  $U_{e'} \subseteq CL_e[p \leftarrow x]$ ,  $CL_K \cup CL_e[p \leftarrow x]$  is unsatisfiable as well. Therefore,  $K \models_k \varphi[p \leftarrow x]$ , and  $\varphi$  is  $p$ -vacuous.  $\square$

**Theorem 4** *Assume  $\varphi$  is  $k$ -step  $p$ -vacuous in  $K$ . Then, there exists a resolution proof  $\Pi$  of unsatisfiability of  $BMC_k(K, \varphi)$  such that  $p$  is peripheral in  $\Pi$ .*

*Proof:* Assume that  $\varphi$  is  $p$ -vacuous. Then, the BMC problem  $BMC_k(K, \varphi[p \leftarrow x]) = CL_K \cup CL_e[p \leftarrow x]$  is unsatisfiable, and there exists a resolution proof  $\Pi$  establishing this. We must show that this proof can be transformed to a proof of unsatisfiability of  $BMC_k(K, \varphi) = CL_K \cup CL_e$  in which each  $p_i$  is peripheral with respect to  $CL_K$  and  $CL_e$ .

If  $\Pi$  does not contain a clause that has both  $p_i$  and  $x_i$  for some  $i$ , then for  $\Pi'$  obtained from  $\Pi$  by replacing each occurrence of  $x_i$  by  $p_i$ , (a)  $\Pi'$  is a well-formed resolution proof, and (b)  $Roots(\Pi') \subseteq CL_K \cup CL_e$ . That is,  $\Pi'$  is a resolution proof establishing that  $BMC_k(K, \varphi)$  is unsatisfiable.

We now show how such a proof can be constructed from an arbitrary proof  $\Pi$  of unsatisfiability of  $BMC_k(K, \varphi[p \leftarrow x])$ . Let  $U_K = Roots(\Pi) \cap CL_K$ , and  $CL_{e'} = CL_e[p \leftarrow x]$ . Then, if  $p_i$  occurs in any clause of  $U_K$ , it is local to  $U_K$ . Let  $L$  be the set of all local variables of  $U_K$ ,  $C = \exists L \cdot U_K$  be a formula resulting from existentially eliminating these local variables, and  $CNF(C)$  be the CNF encoding of  $C$ . For any  $i$ ,  $p_i$  does not appear in  $C$ . Furthermore, the set of clauses  $CNF(C) \cup CL_{e'}$  is unsatisfiable. Thus, there exists a resolution proof  $\Pi'$  establishing this such that  $Roots(\Pi') \subseteq CL_{e'} \cup CNF(C)$ . Finally, since  $U_K \Rightarrow C$ , for each clause  $c \in C$  there exists

a resolution proof  $\Pi_c$  such that  $Leaf(\Pi_c) = c$  and  $Roots(\Pi_c) \subseteq U_K$ . By combining the proofs  $\{\Pi_c \mid c \in CNF(C)\}$  and  $\Pi'$ , we obtain a proof of unsatisfiability of  $U_K \cup CL_{e'}$  that does not contain a clause with variables  $x_i$  and  $p_i$ .  $\square$

## B. Experiments

Table I shows detailed results of our experiments. In this table, column ‘‘Benchmark’’ indicates the benchmark the test case belongs to; ‘‘Test case’’ is the case’s unique identifier inside the benchmark, ‘‘Model’’ is the SMV model tested; ‘‘# var. in  $M$ ’’ is the number of variables in the model; ‘‘ $k$ ’’ is the number of steps used to run BMC; ‘‘op. in  $\varphi$ ’’ shows the property operators (e.g.,  $2G$  means that two G operators appear in the property); ‘‘# var. in  $\varphi$ ’’ is the number of atomic variables present in the property; ‘‘# vac. vars.’’ is the number of vacuous variables; and ‘‘# resol. in  $\Pi$ ’’ is the number of resolutions in the resolution proof. The next three columns report the time needed for model-checking: ‘‘Gen. CNF’’ is the time NuSMV took to generate the corresponding CNF theory; ‘‘Test SAT’’ and ‘‘Gen.  $\Pi$ ’’ are the time MiniSat took to test satisfiability and generate the corresponding resolution proof respectively; and ‘‘Total’’ is the sum of the previous three columns.

For the naive method, we report the total times for the CNF theory generation (‘‘Gen. CNF’’) and for satisfiability testing (‘‘Test SAT’’). One CNF theory is produced per each atomic variable. For irrelevance, local irrelevance and peripherality, we report how many vacuous variables were found by the partial pass (‘‘# vac. vars. found’’), how long *VaqTree* took to do the corresponding analysis (‘‘Anal.’’) and how much time was needed to do the completing pass (‘‘Extra runs’’).

For example, test case **8** analyzes a five-variable, two temporal operator (G,U) property of the **pci** model (which has 40 variables). Only three of these variables are vacuous. The resolution proof generated when  $k = 13$  has 4,283 resolutions. This property was checked in 5.59 seconds. Naive vacuity detection required five model-checking runs, taking 25.85 seconds to generate the corresponding CNF theories and 2.89 seconds to test their satisfiability, requiring a total of 28.74 seconds. Irrelevance took 0.27 seconds to find two of the vacuous variables during the partial pass. It then took 17.60 seconds to carry out the completing pass, so the total time required by irrelevance to find all three vacuous variables is 17.87 seconds. Local irrelevance took 0.28 seconds to analyze the resolution proof, finding the same two vacuous variables as irrelevance. Thus, it also takes 17.60 seconds to run the completing pass, so the total time required by local irrelevance is 17.88 seconds. Finally, peripherality took 0.47 seconds to execute the partial pass and found the same two vacuous variables; it also required 17.60 seconds to run the completing pass, taking a total of 18.07 seconds to produce complete results for test case **8**.

*VaqTree*, the complete experimental results and some test cases are available at [19].

TABLE I: Statistics for vacuity detection experiments on NuSMV distribution and other examples.

Bench- mark	Test case	Model ( $M$ )	# var. in $M$	k	op. in $\varphi$	# var. in $\varphi$	# vac. vars.	# resol. in $\Pi$	Model Checking				Naive			Irrelevance			Local Irrelevance (LI)			Peripherality (P)					
									Gen.	Test	Gen.	Total	Gen.	Test	Total	# vac. vars. found	Anal.	Extra	Total	# vac. vars. found	Anal.	Extra	Total	# vac. vars. found	Anal.	Extra	Total
									CNF (s)	SAT (s)	$\Pi$ (s)	(s)	CNF (s)	SAT (s)	(s)	(s)	(s)	runs (s)	(s)	(s)	(s)	runs (s)	(s)	(s)	(s)	runs (s)	(s)
$S_A$	1	pci	40	13	G,U	4	1	19792	4.69	0.23	5.9	10.82	20.66	2.77	23.43	0	0.34	23.43	23.77	0	0.34	23.43	23.77	0	0.81	23.43	24.24
$S_A$	2	pci	40	13	G,U	4	3	1649	5.13	0.14	5.64	10.91	11.75	1.30	13.05	3	0.26	0	0.26	3	0.26	0	0.26	3	0.37	0	0.37
$S_A$	3	pci	40	13	G,U	4	3	1649	5.09	0.13	5.32	10.54	12.03	2.14	14.17	3	0.26	0	0.26	3	0.25	0	0.25	3	0.37	0	0.37
$S_A$	4	pci	40	13	G,U	3	1	7725	4.80	0.18	5.65	10.63	12.68	1.73	14.41	0	0.29	14.41	14.7	0	0.29	14.41	14.70	0	0.50	14.41	14.91
$S_A$	5	pci	40	13	G,U	3	1	7555	4.76	0.18	5.55	10.49	12.36	1.56	13.92	0	0.28	13.92	14.20	0	0.28	13.92	14.20	0	0.50	13.92	14.42
$S_A$	6	pci	40	13	G,U	4	3	1705	4.66	0.12	5.68	10.46	11.66	1.19	12.85	3	0.25	0	0.25	3	0.26	0	0.26	3	0.39	0	0.39
$S_A$	7	pci	40	13	G,U	4	3	1705	4.67	0.14	5.42	10.23	11.68	1.40	13.08	3	0.25	0	0.25	3	0.26	0	0.26	3	0.37	0	0.37
$S_A$	8	pci	40	13	G,U	5	3	4283	4.95	0.22	5.59	10.76	25.85	2.89	28.74	2	0.27	17.60	17.87	2	0.28	17.60	17.88	2	0.47	17.60	18.07
$S_A$	66	prod-cell	39	10	G,F	6	6	5275	0.88	0.04	1.25	2.17	5.55	0.18	5.73	6	0.28	0	0.28	6	0.28	0	0.28	6	0.49	0	0.49
$S_A$	67	prod-cell	39	10	G,F	5	5	5320	1.02	0.04	1.41	2.47	4.81	0.16	4.97	5	0.28	0	0.28	5	0.29	0	0.29	5	0.47	0	0.47
$S_A$	68	prod-cell	39	10	G,F	4	4	3798	0.91	0.03	1.27	2.21	3.57	0.12	3.69	2	0.27	1.86	2.13	2	0.27	1.86	2.13	2	0.43	1.86	2.29
$S_A$	69	prod-cell	39	10	G,F	8	8	2764	0.99	0.03	1.26	2.28	7.52	0.23	7.75	1	0.26	6.78	7.04	1	0.26	6.78	7.04	1	0.42	6.78	7.2
$S_A$	70	prod-cell	39	10	G,F	5	5	5232	1.20	0.04	1.33	2.57	4.63	0.15	4.78	1	0.28	3.82	4.10	2	0.28	2.86	3.14	2	0.48	2.86	3.34
$S_A$	71	prod-cell	39	10	G,F	4	4	4068	1.35	0.03	1.27	2.65	3.87	0.10	3.97	2	0.27	2.16	2.43	3	0.27	0.95	1.22	3	0.44	0.95	1.39
$S_A$	72	prod-cell	39	10	G,F	4	4	2756	0.96	0.03	1.27	2.26	3.64	0.13	3.77	1	0.26	2.82	3.08	1	0.26	2.82	3.08	1	0.40	2.82	3.22
$S_A$	73	prod-cell	39	10	G,F	6	6	4425	0.84	0.04	1.30	2.18	5.47	0.19	5.66	2	0.28	3.74	4.02	2	0.28	3.74	4.02	2	0.46	3.74	4.2
$S_A$	74	prod-cell	39	10	G,F	5	5	3802	0.92	0.04	1.28	2.24	4.55	0.17	4.72	4	0.27	1.01	1.28	4	0.28	1.01	1.29	4	0.43	1.01	1.44
$S_A$	75	prod-cell	39	10	G,F	5	5	2802	0.91	0.03	1.44	2.38	4.53	0.14	4.67	2	0.26	2.80	3.06	2	0.26	2.80	3.06	2	0.41	2.80	3.21
$S_A$	76	prod-cell	39	10	G,F	8	8	3732	1.16	0.03	1.36	2.55	7.72	0.21	7.93	5	0.28	2.96	3.24	6	0.27	1.98	2.25	6	0.46	1.98	2.44
$S_A$	77	prod-cell	39	10	G,F	9	9	3010	1.50	0.03	1.28	2.81	8.93	0.22	9.15	6	0.27	3.12	3.39	7	0.27	1.94	2.21	7	0.45	1.94	2.39
$S_A$	78	prod-cell	39	10	G,F	5	5	2585	0.86	0.03	1.25	2.14	4.98	0.14	5.12	2	0.26	2.93	3.19	2	0.26	2.93	3.19	2	0.40	2.93	3.33
$S_A$	79	prod-cell	39	10	G,F	5	5	2556	1.06	0.03	1.30	2.39	4.70	0.12	4.82	2	0.26	2.98	3.24	2	0.26	2.98	3.24	2	0.40	2.98	3.38
$S_A$	80	prod-cell	39	10	G,F	4	4	5317	1.26	0.04	1.27	2.57	3.53	0.12	3.65	4	0.29	0	0.29	4	0.29	0	0.29	4	0.46	0	0.46
$S_A$	81	prod-cell	39	10	G,2F	10	10	2497	3.15	0.06	1.29	4.5	9.68	0.27	9.95	3	0.26	6.97	7.23	4	0.26	4.94	5.20	4	0.42	4.94	5.36
$S_A$	82	prod-cell	39	10	G,F	8	8	2348	0.88	0.033	1.25	2.16	7.52	0.22	7.74	3	0.27	4.84	5.11	3	0.26	4.84	5.10	3	0.41	4.84	5.25
$S_A$	83	abp4	13	19	G,F	1	0	1289374	2.79	10.73	34.14	47.66	2.93	1.79	4.72	0	5.51	4.72	10.23	0	5.72	4.72	10.44	0	98.62	4.72	103.34
$S_A$	84	abp4	13	19	G,F	3	2	1050234	3.14	6.45	29.43	39.02	8.43	20.76	29.19	0	5.07	29.19	34.26	0	5.22	29.19	34.41	0	67.54	29.19	96.73
$S_A$	85	abp4	13	19	G,F	3	2	2246095	2.99	19.03	49.63	71.65	8.81	26.43	35.24	0	8.23	33.78	42.01	0	8.22	33.78	42.01	0	412.30	33.78	446.08
$S_A$	86	abp4	13	19	G,2F	2	0	795705	3.07	5.04	21.28	29.39	5.54	6.29	11.83	0	2.69	25.64	28.33	0	2.71	25.64	28.35	0	37.21	25.64	62.85
$S_A$	93	toyFGS04	151	18	F	6	6	297	18.88	0.26	5.27	24.41	114.78	0.76	115.54	3	0.23	57.39	57.62	3	0.22	57.39	57.61	3	0.29	57.39	57.68
$S_A$	94	toyFGS04	151	18	F	12	12	308	19.13	0.16	5.28	24.57	224.79	1.40	226.19	6	0.26	132.43	132.69	6	0.26	132.43	132.69	6	0.33	132.43	132.76
$S_A$	95	toyFGS04	151	18	F	6	0	318	18.35	0.15	5.17	23.67	126.28	32.03	158.31	0	0.22	158.31	158.53	0	0.22	158.31	158.53	0	0.29	158.31	158.60
$S_A$	96	toyFGS04	151	18	F	4	0	308	18.57	0.14	5.45	24.16	75.18	22.26	97.44	0	0.22	97.44	97.66	0	0.22	97.44	97.66	0	0.27	97.44	97.71
$S_A$	97	toyFGS04	151	18	G	4	0	8072	14.14	0.21	3.3	17.65	57.91	10.60	68.51	0	0.33	68.51	68.84	0	0.33	68.51	68.84	0	0.60	68.51	69.11
$S_A$	98	toyFGS04	151	18	G	6	0	7985	14.47	0.21	3.63	18.31	88.94	11.48	100.42	0	0.34	100.42	100.76	0	0.34	100.42	100.76	0	0.68	100.42	101.10
$S_A$	99	toyFGS04	151	18	F	6	6	293	19.80	0.15	5.61	25.56	111.91	0.66	112.57	2	0.21	75.08	75.29	2	0.22	75.08	75.30	2	0.27	75.08	75.35
$S_A$	107	msi_wtrans	30	40	G	5	3	66	21.85	0.20	8.39	30.44	120.15	65.70	185.85	3	0.21	112.59	112.80	3	0.20	112.59	112.79	3	0.24	112.59	112.83
$S_A$	108	msi_wtrans	30	40	F	5	4	66	23.53	0.20	9.15	32.88	120.16	73.28	193.44	3	0.2	120.30	120.50	3	0.21	120.30	120.51	3	0.25	120.30	120.55
$S_A$	109	msi_wtrans	30	40	F	6	4	66	21.56	0.21	8.46	30.23	156.61	93.23	249.84	4	0.21	0	0.21	4	0.21	0	0.21	4	0.24	0	0.24
$S_B$	1	IBM_FV_2002_03	111	20	G	8	8	7480	4.54	0.09	3.8	8.43	36.21	0.67	36.88	7	0.35	4.67	5.02	7	0.35	4.67	5.02	7	0.74	4.67	5.41
$S_B$	2	IBM_FV_2002_04	223	20	G	4	3	45065	7.62	0.92	5.71	14.25	29.66	3.83	33.49	0	0.59	33.49	34.08	0	0.59	33.49	34.08	0	1.67	33.49	35.16
$S_B$	3	IBM_FV_2002_05	310	20	G	2	1	32776	11.82	0.62	10.02	22.46	22.97	1.31	24.28	1	0.44	12.21	12.65	1	0.44	12.21	12.65	1	1.02	12.21	13.23
$S_B$	4	IBM_FV_2002_09	233	20	F	9	9	2	8.96	0.17	0	9.13	81.02	1.22	82.24	9	0.17	0	0.17	9	0.17	0	0.17	9	0.17	0	0.17
$S_B$	5	IBM_FV_2002_10	224	20	G	3	2	78523	54.23	8.45	46.09	108.77	165.88	93.22	259.1	0	0.7	259.1	259.8	0	0.7	259.1	259.8	0	2.33	259.1	261.43
$S_B$	6	IBM_FV_2002_10	224	20	G	4	3	177536	53.3	30.21	56.61	140.12	219.74	199.25	418.99	0	1.12	418.99	420.11	0	1.12	418.99	420.11	0	5.8	418.99	424.79
$S_B$	7	IBM_FV_2002_10	224	20	G	4	4	9119	53.97	0.97	40.84	95.78	218.45	211.21	429.66	3	0.32	112.58	112.9	3	0.32	112.58	112.9	3	0.61	112.58	113.19
$S_B$	8	IBM_FV_2002_10	224	20	G	2	0	155775	54.99	9.22	46.75	110.96	108.76	165.12	273.86	0	0.99	273.86	274.85	0	0.99	273.86	274.85	0	3.7	273.86	277.56
$S_B$	9	IBM_FV_2002_10	224	20	G	2	1	197053	54.96	65.43	79.32	199.71	110.2	103.82	214.02	0	1.09	214.02	215.11	0	1.12	214.02	215.14	0	4.5	214.02	218.52
$S_B$	10	IBM_FV_2002_17_1	1584	20	G	2	0	96085	38.58	1.15	14.23	53.96	75.78	2.35	78.13	0	1.07	78.13	79.2	0	1.08	78.13	79.2	0			

# Improved Design Debugging using Maximum Satisfiability

Sean Safarpour, Hratch Mangassarian, Andreas Veneris  
Department of Elec. and Comp. Eng.  
University of Toronto, Toronto, Canada  
{sean, hratch, veneris}@eecg.toronto.edu

Mark H. Liffiton, Karem A. Sakallah  
Department of Elec. Eng. and Comp. Sci.  
University of Michigan, Ann Arbor, USA  
{liffiton, karem}@eecs.umich.edu

**Abstract**—In today’s SoC design cycles, debugging is one of the most time consuming manual tasks. CAD solutions strive to reduce the inefficiency of debugging by identifying error sources in designs automatically. Unfortunately, the capacity and performance of such automated techniques must be considerably extended for industrial applicability. This work aims to improve the performance of current state-of-the-art debugging techniques, thus making them more practical. More specifically, this work proposes a novel design debugging formulation based on maximum satisfiability (max-sat) and approximate max-sat. The developed technique can quickly discard many potential error sources in designs, thus drastically reducing the size of the problem passed to an existing debugger. The max-sat formulation is used as a pre-processing step to construct a highly optimized debugging framework. Empirical results demonstrate the effectiveness of the proposed framework as run-time improvements of orders of magnitude are consistently realized over a state-of-the-art debugger.

## I. INTRODUCTION

Functional verification tasks dominate the effort of contemporary VLSI and SoC design cycles. A major step of functional verification is design debugging, which determines the root cause of failed verification tasks such as simulation or equivalence checking. For example, when a simulation run fails because a design’s behavior is inconsistent with its specification, debugging identifies the components responsible for the discrepancy.

Hardware debugging is overwhelmingly performed manually in the industry today. Designers and verification engineers must analyze the failed verification instances, the design and the specification to realize which design components or blocks are the root cause of the failure. Due to the “guess-and-check” nature of the problem, this task is accepted as one of the most time-consuming processes of the VLSI and SoC design cycles. As design complexities nearly double with every generation, so does the daunting debugging effort. Clearly, automated debugging solutions are needed to increase a designer’s debugging and verification efficiency.

Automated debugging is a computationally intensive problem since its complexity increases dramatically with the size of the design, the length of the error traces and with the number of errors present in the design. There is a rich history of debugging techniques and algorithms developed over the last decades which seek to tackle this problem [1], [2], [6]. Although efficient for relatively small design blocks and particular design types, these solutions have not been

extended to industrial problems. More recently, several debugging techniques based on formal techniques such as Boolean Satisfiability (SAT) [10] and Quantified Boolean Formula (QBF) [5] have demonstrated great promise and encouraged further research in formal techniques for debugging. Despite these successes, the capacity and performance of both traditional and newer debugging techniques must be greatly improved to make debugging practical for industrial problems.

This work proposes a novel framework with the aim of greatly reducing the run-time of state-of-the-art debuggers. This technique presents the first maximum satisfiability (max-sat) formulation for design debugging. The formulation is constructed using the constraints corresponding to the erroneous design, the input stimulus, and the expected correct response. The formulation is unsatisfiable, since the incorrect design cannot produce the correct response, and it can only be satisfied if some of the constraints are removed. An all-solution max-sat solver can iteratively find maximal satisfiable subsets of the constraints. The complement of any of these subsets is a set of constraints whose removal makes the problem satisfiable. These constraints will be shown to correspond directly to the erroneous gates or components in the design. The proposed max-sat technique is developed for combinational and sequential circuits as well as for problems with single or multiple input stimuli and expected responses.

The proposed technique is an alternative approach to hardware debugging which can be easily enhanced to over-approximate solutions. The over-approximation allows for a trade-off between the tool’s performance and the resolution of the solutions. More specifically, approximation can reduce the problem complexity and thus require less run-time at the cost of finding larger, less precise solutions. Although not exact, this approach can be employed as a pre-processing step that filters solutions for a second stage exact debugger. The second debugger benefits from having fewer suspect error sources which translates into faster run-times. The combined two-step debugging framework reduces the complexity of both stages, resulting in an efficient debugging solution.

A suite of experiments on combinational and sequential circuits for single and multiple vectors are conducted to demonstrate the benefit of the proposed framework. On average, the over-approximation technique quickly eliminates 92% of the suspects. The second stage debugger uses the filtered suspects to find the exact error sources in a fraction of the time

it would take otherwise. Overall, performance improvements of 200 times or two orders of magnitude over a state-of-the-art debugger are observed consistently.

In the next section, background is provided on the max-sat approach used as well as on design debugging. Sec. III presents the proposed max-sat approach for combinational circuits and Sec. IV extends this for sequential circuits and for multiple vectors. Sec. V present the over-approximation technique and the overall framework developed for optimal performance, respectively. Experiments are presented in Sec. VI followed by the conclusion in Sec. VII.

## II. BACKGROUND

### A. Maximum Satisfiability

The algorithm from [8] is used to solve a generalization of the max-sat problem. While max-sat is concerned with finding a satisfiable set of clauses with maximum cardinality, this can be generalized to find Maximal Satisfiable Subsets (MSSes). An MSS is a satisfiable subset of a formula's clauses that is maximal in the sense that adding any one of the remaining clauses would make it UNSAT. Any max-sat solution is of course an MSS, but MSSes can be different (smaller) sizes as well. In this work, the *complements* of MSSes, sets of clauses whose removal makes the instance satisfiable, are of interest. Just as an MSS is maximal, its complement is minimal, and we refer to such a set as a Minimal Correction Set (MCS). This work makes use of two following techniques developed as extensions to the algorithm from [8]:

- Finding all MCSes up to size  $k$
- Grouping clauses to produce "approximate" MCSes

Finding all MCSes up to size  $k$  is performed by the algorithm AllMCSes from [8], which was developed as the first phase of an approach for finding all Minimal Unsatisfiable Subsets (MUSes). This procedure solves consecutive optimization problems, finding MCSes in order of increasing size (equivalent to finding their complementary MSSes in order of decreasing size). MCSes are returned as they are found, and execution can be stopped when a size limit is reached.

The second ability, of grouping clauses, depends on the way the algorithm uses clause-selector variables. Every clause  $C_i$  is augmented with a new variable  $y_i$ , producing  $C'_i = (\bar{y}_i + C_i) = (y_i \rightarrow C_i)$ . When  $y_i$  is assigned TRUE, the original clause  $C_i$  must be satisfied, while when  $y_i$  is FALSE,  $C'_i$  is satisfied, essentially disabling the original clause. This gives a standard SAT solver the ability to enable and disable constraints implicitly within the normal backtracking search. By assigning the same  $y$  variable to *multiple* clauses, a set of clauses can be treated as a single higher-level constraint (the conjunction of all clauses given the same  $y$  variable) that can be enabled and disabled at once. Using this approach, each MCS is a minimal set of *groups* of constraints whose removal makes the instance satisfiable. This leads to an over-approximation of an MCS of the original clauses, because extra clauses will be included in groups even though they may not be necessary. The benefit of the over-approximation is that

it can greatly increase the performance of the algorithm as the search space is reduced exponentially.

This work uses the MCS techniques outlined above for debugging. Although not precise in the general case, the term max-sat is used throughout to refer collectively to the techniques above for simplicity.

### B. Automated Design Debugging

The problems of *design debugging* and *fault diagnosis*, which occur at different stages of the VLSI design cycle, have strong similarities. The latter occurs when a fabricated chip fails during the testing phase due to the presence of manufacturing defects [10], while design debugging occurs at the early stages of the design cycle, when the implemented design does not meet its functional specifications. In this paper the terminology and assumptions are those of design debugging, however, the proposed techniques can apply to fault diagnosis as well.

The input of the design debugging problem is an erroneous circuit  $C$ , a set of input stimuli  $I$  for which the design fails verification, and the corresponding correct output responses  $O$ . The components  $I$  and  $O$ , also called input/output vectors, can be obtained from simulation-based verification tools or formal tools such as equivalence checkers and model checkers.

An error source at the circuit-level exhibits an erroneous response at the primary outputs for at least one of the provided vectors. In this paper, a model-free diagnosis strategy is used, which can "detect" any type of gate/module error. The *error cardinality*  $N_g$  is the maximum number of simultaneous error sources the debugger assumes exist in the circuit. The complexity of design debugging increases exponentially with the error cardinality [11]. A design debugging tool must return all possible solutions, *i.e.* all potential error tuples up to the size of the error cardinality.

Traditionally, methods based on simulation, path-tracing and binary decision diagrams have been used to tackle the design debugging problem [1], [2], [6]. Recently, SAT-based strategies [10] have been proven to be effective as their performance increases with that of the underlying SAT solvers. This approach formulates the design debugging problem by constructing circuit constraints, translating it to a Boolean formula in *Conjunctive Normal Form* (CNF), and giving it to an all-solution SAT solver. Note that deriving a CNF from a circuit is a simple linear time algorithm as there is a one-to-one correspondence between circuit gates and CNF formulas. Table I shows five basic gates along with their CNF representations.

Gate	CNF
$y = \text{NOT}(x)$	$(x + y) \cdot (\bar{x} + \bar{y})$
$y = \text{AND}(x_1, x_2, \dots, x_n)$	$(x_1 + \bar{y}) \cdot (x_2 + \bar{y}) \cdot \dots \cdot (x_n + \bar{y}) \cdot (\bar{x}_1 + \bar{x}_2 + \dots + \bar{x}_n + y)$
$y = \text{OR}(x_1, x_2, \dots, x_n)$	$(\bar{x}_1 + y) \cdot (\bar{x}_2 + y) \cdot \dots \cdot (\bar{x}_n + y) \cdot (x_1 + x_2 + \dots + x_n + \bar{y})$
$y = \text{XOR}(x_1, x_2)$	$(\bar{x}_1 + x_2 + y) \cdot (x_1 + \bar{x}_2 + y) \cdot (x_1 + x_2 + \bar{y}) \cdot (\bar{x}_1 + \bar{x}_2 + \bar{y})$
$y = \text{MUX}(s, x_1, x_2)$	$(x_1 + \bar{y} + s) \cdot (\bar{x}_1 + y + s) \cdot (x_2 + \bar{y} + \bar{s}) \cdot (\bar{x}_2 + y + \bar{s})$

TABLE I  
GATE TO CNF TRANSLATION

### III. DEBUGGING COMBINATIONAL CIRCUITS WITH MAX-SAT

Given an erroneous circuit  $C$ , an input stimulus  $I$ , and the corresponding correct output response  $O$  a CNF formula can be produced as follows.

$$\Phi = I \cdot O \cdot \text{CNF}(C)$$

This CNF problem is naturally unsatisfiable because the erroneous circuit cannot produce the correct output response under the given input vector. Since the inconsistency between a circuit's actual and correct response is due to some gate-level error sources, the unsatisfiability of the problem is due to the clauses derived from these error sources. In other words, the clauses that are at conflict in the CNF correspond to the circuit-level error sources from which they are derived. Therefore, the circuit-level errors can be identified by finding the CNF-level *error clauses*.

The max-sat approach in Sec. II-A can identify Maximal Satisfiable Subsets (MSSes) whose complements are Minimal Correction Sets (MCSes). These MCSes represent sets of clauses whose removal from the CNF make the problem satisfiable. In the formula  $\Phi$  constructed using the constraints  $I$ ,  $O$ , and  $\text{CNF}(C)$ , the MCSes map directly to error clauses. Once the error clauses are identified through MCSes, the gate-level suspects are found by mapping each clause to the gate it is originally derived from as described in Sec. II.

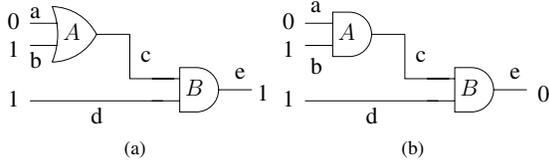


Fig. 1. Correct and erroneous circuit

For example, consider the correct and erroneous circuit in Fig. 1 (a) and (b) where gate  $A$  is mistakenly implemented as an AND gate instead of an OR gate. Under the input stimulus  $\{a = 0, b = 1, d = 1\}$  the circuit has a response of  $\{e = 0\}$  instead of the correct response of  $\{e = 1\}$ . The corresponding erroneous CNF for the circuit and the input/output vectors are shown below.

$$\begin{aligned} & (\bar{a}) \cdot (b) \cdot (d) \cdot (e) \\ & (a + \bar{c}) \cdot (b + \bar{c}) \cdot (\bar{a} + \bar{b} + c) \\ & (c + \bar{e}) \cdot (d + \bar{e}) \cdot (\bar{c} + \bar{d} + e). \end{aligned}$$

Here, the max-sat approach described in Sec. II-A can return the MCS  $(a + \bar{c})$  as a solution because removing this clause from the CNF makes the formula satisfiable. Notice that this clause is derived from the erroneous gate  $A$ .

The above example illustrates how the removal of an error clause can help identify the error source. Further analysis of the example demonstrates that there are other clauses such as  $(c + \bar{e})$  whose removal can satisfy the problem. Indeed, more than one error clause may exist in a given problem

corresponding to the many potential error sources at the gate-level. These are more commonly known as equivalent errors or faults in the diagnosis literature [1]. Note that the removal of the clause  $(\bar{a})$  also satisfies the problem, however since this constraint is not part of the circuit component of the CNF (*i.e.*  $C$ ), it is not considered as an error clause.

For the debugging technique to be complete, all equivalent errors must be found. Each of these is known as a suspect error source because it may fix the problem such that erroneous circuit produces the correct response for the given input vector. As a result, the AllMCSes algorithm of Sec. II-A is used to find all error clauses and consequently all gate-level error suspects.

#### A. Error Clause Cardinality

Since the solution space for the AllMCSes algorithm is exponential, an explicit limit for the maximum cardinality of the MCSes is advised to prevent memory explosion. In practice, this limit, called the *error clause cardinality*, must be relatively small due to memory and performance considerations. The error clause cardinality determines the completeness and efficiency of the proposed technique.

Since this work is primarily concerned with gate-level debugging the limit used must correspond with the gate-level cardinality of conventional debuggers. In Sec. II the error cardinality  $N_g$  is defined as the maximum gate tuples that may be responsible for the erroneous behavior. At the level of the CNF encoding, the error clause cardinality  $N_c$  must be set to a value such that all the gate-level errors at  $N_g$  can be found using the proposed max-sat approach. Thus completeness in this context is with respect to the gate-level debuggers such as [5]. The following theorem proves that the proposed approach is complete for a given value of  $N_g$ .

**Theorem:** The algorithm AllMCSes called on the problem  $\Phi = I \cdot O \cdot \text{CNF}(C)$  with a limit of  $N_c$  is complete if  $N_c$  is equal to [the maximum number of clauses derived for any single gate in the CNF]  $\times N_g$ .

**Proof:** Proof by contradiction. Suppose there is a gate-level error not identified by the proposed approach using the error cardinality limit  $N_c$ . Since AllMCSes iteratively finds sets of clauses with cardinality 1 up to  $N_c$ , the gate-level error must be caused by more than  $N_c$  clauses. However,  $N_c$  is equal to the maximum number of clauses derived from any one gate times  $N_g$ , so the error must be caused by more than  $N_g$  gate-level sources. Therefore the error is not found using conventional debuggers with  $N_g$  either.  $\square$

In many circuit-based SAT problems, the circuit is first converted to a 2-input AND-INVERTER graph and then translated into CNF [4], [7]. In such a CNF formula, the maximum number of clauses from any gate is 3, thus  $N_c = 3 \times N_g$ . Using this value for  $N_c$  results in finding all the solutions found using conventional debuggers with  $N_g$ . In CNF formulas derived from arbitrary circuits where the number of clauses generated can greatly vary from one gate to another, the proposed max-sat debugging technique may return more solutions than the gate-level debugger for a given  $N_g$ . As discussed further

in Sec. V this scenario does not pose a problem under the proposed framework.

### B. Error Group Cardinality

The previous section presented a limit for the error clause cardinality to guarantee completeness for the proposed approach. Although complete, increasing the error clause cardinality is not always desired as the complexity of the debugging problem is exponentially related to the error cardinality. Here, the grouping ability described in Sec. II-A is used to reduce the complexity of the problem while maintaining completeness.

Grouping all clauses derived from the same gate together allows the max-sat solver to “enable” or “disable” all of those clauses simultaneously. In effect, this gives the solver the ability to treat each gate as a single high-level constraint, leading to solutions (MCSes) found directly in terms of the gates. Under this problem restriction, the *error clause-group cardinality*,  $N_{cg}$  required to find gate-level errors can be effectively  $N_g$ .

**Theorem:** By grouping all clauses derived from the same gate together, the proposed technique is complete if the error clause-group cardinality  $N_{cg} = N_g$ .

**Proof:** Since each group has a one-to-one correspondence with a circuit gate, when a group is found as part of an MCS, all clauses corresponding to the original gate are “disabled” by the AllMCSes algorithm. Thus every solution found by AllMCSes maps to a set of the original gates. Hence, limiting the group cardinality is equivalent to limiting the gate cardinality.  $\square$

Re-visiting the example of Fig. 1, grouping the clauses of gate  $A$  together with the clause-selector variable  $y_A$  and the clauses of gate  $B$  together with the clauses-selector variable  $y_B$ , results in the following CNF.

$$\begin{aligned} & (\bar{a}) \cdot (b) \cdot (d) \cdot (e) \\ & (a + \bar{c} + \bar{y}_A) \cdot (b + \bar{c} + \bar{y}_A) \cdot (\bar{a} + \bar{b} + c + \bar{y}_A) \\ & (c + \bar{e} + \bar{y}_B) \cdot (d + \bar{e} + \bar{y}_B) \cdot (\bar{c} + \bar{d} + e + \bar{y}_B). \end{aligned}$$

## IV. EXTENSION TO SEQUENTIAL CIRCUITS AND MULTIPLE VECTORS

Debugging sequential circuits is similar to that of combinational circuits except that their behavior must be modeled for a finite number of clock cycles. These clock cycles are necessary to excite and observe the errors. A popular approach for modeling sequential circuits is to use the time-frame expansion technique or the Iterative Logic Array (ILA) representation. These techniques replicate a circuit’s transition relation, called a time-frame, and connect the current-state and the next-state of adjacent time-frames together. In effect, the sequential circuit is transformed into an “unfolded” combinational circuit that can be debugged like any other combinational circuit.

Since the complexity of debugging increases exponentially with the number of error sources, debuggers must be careful not to consider the “replicated” gates across time-frames as unique error sources. For example, a single gate-level error in an ILA with 3 time-frames may appear to have 3 distinct error locations, however, replacing the functionality of a single gate

in the original sequential circuit will fix the problem in all time-frames.

The proposed max-sat debugging technique can be extended to handle sequential designs efficiently. First, the sequential circuit is converted to an ILA and then translated into CNF. Similar to the previous formulation the CNF is then constrained with input stimulus and output response,  $I$  and  $O$  resulting in

$$\Phi = I \cdot O \cdot \text{CNF}(\text{ILA}(C)).$$

The second step is to account for the replication due to the ILA by grouping all clauses derived from the same gate but from any time-frame. As a result, clauses from a particular gate will be “enabled” and “disabled” at once irrespective of the time-frames they represent.

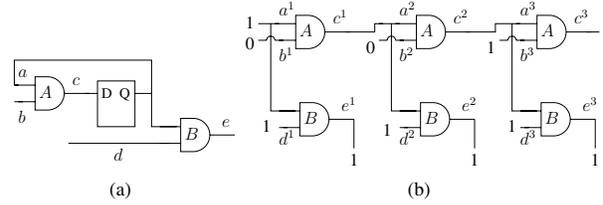


Fig. 2. Erroneous sequential circuit and its ILA representation

For example consider the erroneous sequential circuit shown in Fig. 2(a) and its ILA in Fig. 2(b). Here, the gate  $A$  has been erroneously implemented as an AND gate instead of an OR gate. As a result, the output of  $A$  in the first and second time-frames should be 1 instead of 0. Note that the input stimulus and correct response are also shown in Fig. 2(b). The corresponding CNF for the constrained ILA is shown below.

$$\begin{aligned} & (a^1) \cdot (\bar{b}^1) \cdot (d^1) \cdot (e^1) \\ & (a^1 + \bar{c}^1) \cdot (b^1 + \bar{c}^1) \cdot (\bar{a}^1 + \bar{b}^1 + c^1) \\ & (a^1 + \bar{e}^1) \cdot (d^1 + \bar{e}^1) \cdot (\bar{a}^1 + \bar{d}^1 + e^1) \\ & (\bar{c}^1 + a^2) \cdot (c^1 + \bar{a}^2) \\ & (\bar{b}^2) \cdot (d^2) \cdot (e^2) \\ & (a^2 + \bar{c}^2) \cdot (b^2 + \bar{c}^2) \cdot (\bar{a}^2 + \bar{b}^2 + c^2) \\ & (a^2 + \bar{e}^2) \cdot (d^2 + \bar{e}^2) \cdot (\bar{a}^2 + \bar{d}^2 + e^2) \\ & (\bar{c}^2 + a^3) \cdot (c^2 + \bar{a}^3) \\ & (\bar{b}^3) \cdot (d^3) \cdot (e^3) \\ & (a^3 + \bar{c}^3) \cdot (b^3 + \bar{c}^3) \cdot (\bar{a}^3 + \bar{b}^3 + c^3) \\ & (a^3 + \bar{e}^3) \cdot (d^3 + \bar{e}^3) \cdot (\bar{a}^3 + \bar{d}^3 + e^3) \end{aligned}$$

In the above example, the clauses corresponding to gate  $A$  in both time frames 1 and 2 are responsible for the discrepancy between the actual and correct response. Specifically, these are  $(b^1 + \bar{c}^1)$  and  $(b^2 + \bar{c}^2)$ . However, by grouping all clauses derived from gate  $A$  together and those from gate  $B$  together, irrespective of the time-frames, the single group solution is returned. Below is the modified CNF based on grouping clauses from gate  $A$  ( $B$ ) together with the clause-selector variable  $y_A$  ( $y_B$ ).

$$\begin{aligned}
& \frac{(a^1) \cdot (\overline{b^1}) \cdot (d^1) \cdot (e^1)}{(a^1 + \overline{c^1} + \overline{y_A}) \cdot (b^1 + \overline{c^1} + \overline{y_A}) \cdot (a^1 + \overline{b^1} + c^1 + \overline{y_A})} \\
& \frac{(a^1 + \overline{c^1} + \overline{y_B}) \cdot (d^1 + \overline{e^1} + \overline{y_B}) \cdot (a^1 + \overline{d^1} + e^1 + \overline{y_B})}{(c^1 + a^2) \cdot (c^1 + a^2)} \\
& \frac{(b^2) \cdot (d^2) \cdot (e^2)}{(a^2 + \overline{c^2} + \overline{y_A}) \cdot (b^2 + \overline{c^2} + \overline{y_A}) \cdot (a^2 + \overline{b^2} + c^2 + \overline{y_A})} \\
& \frac{(a^2 + \overline{c^2} + \overline{y_B}) \cdot (d^2 + \overline{e^2} + \overline{y_B}) \cdot (a^2 + \overline{d^2} + e^2 + \overline{y_B})}{(c^2 + a^3) \cdot (c^2 + a^3)} \\
& \frac{(b^3) \cdot (d^3) \cdot (e^3)}{(a^3 + \overline{c^3} + \overline{y_A}) \cdot (b^3 + \overline{c^3} + \overline{y_A}) \cdot (a^3 + \overline{b^3} + c^3 + \overline{y_A})} \\
& \frac{(a^3 + \overline{c^3} + \overline{y_B}) \cdot (d^3 + \overline{e^3} + \overline{y_B}) \cdot (a^3 + \overline{d^3} + e^3 + \overline{y_B})}{(a^3 + \overline{c^3} + \overline{y_B}) \cdot (d^3 + \overline{e^3} + \overline{y_B}) \cdot (a^3 + \overline{d^3} + e^3 + \overline{y_B})}
\end{aligned}$$

For debugging problems with multiple vectors,  $\vec{I} = \{I_1, I_2, \dots\}$ ,  $\vec{O} = \{O_1, O_2, \dots\}$ , the union of the CNF problems for each vector results in a single constraint system. In other words the CNF corresponding to the circuit,  $C$  is again replicated for each vector. Similar to the approach for sequential circuit, all clauses derived from the same gate, regardless of which replica of  $C$  they occur in, must be grouped together and treated as a single error source. It should be noted that the groupings for multiple vectors and sequential circuits is in addition to the gate groupings discussed in Sec. III.

## V. DEBUGGING WITH APPROXIMATE MAX-SAT

In practice, debugging via an exact max-sat formulation may not be practical, as the number of groups and clauses under consideration can be quite high thus resulting in a “hard” max-sat problem. The proposed max-sat strategy can be easily modified to perform an over-approximation instead of finding exact solutions. The benefit of the over-approximation is that the speed and resolution trade-off can be adjusted for the problem: reducing the resolution or granularity of the solutions found yields decreased run-time.

The over-approximation is achieved by grouping clauses together as described in Sec. II-A and finding the MCSes in terms of the groups. Note that the groupings discussed here are in addition to those presented in Sec. III and IV. Different grouping strategies can be easily formulated ranging from random groupings to those based on a circuit’s topology or structure. Similarly, groups can differ in cardinality from a single clause to thousands of clauses. For instance, a set of clauses can be grouped together if they are in the same fanout-free cone which is similar to the dominator debugging technique introduced in [10]. Another example is grouping based on a high-level modules derived from RTL similar to the technique of [3]. Intuitively, generating groups based on the circuit’s structure or modularity may be advantageous as fewer solutions/suspects may be returned compared to arbitrary grouping schemes.

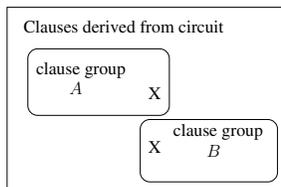


Fig. 3. Error masking in clause groupings

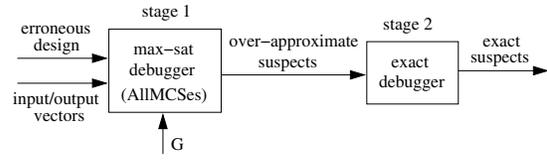


Fig. 4. Max-sat debugging framework

Grouping clauses may increase the effect of error masking, in which some error sources may not be detected as they are masked by others [3]. This also occurs in traditional diagnosis techniques when error-free models are used. For instance, consider the gates shown in Fig. 1 and a pair of errors on gates  $A$  and  $B$ . In this scenario, the single model-free error,  $A$ , masks the pair solution of  $A$  and  $B$ .

Similar scenarios can occur when grouping clauses together, especially if the groups are made arbitrarily. For instance, consider the CNF illustrated in Fig. 3 where some clauses are grouped in  $A$  and other are grouped in  $B$ . Further consider a pair of error clauses illustrated by the “X”. Here, the single solution identifying group  $A$  masks the pair solution  $A$  and  $B$ . It should be emphasized that error masking is not unique to the proposed technique as it occurs in gate-level and hierarchical debugging as well [3].

### A. Efficient Max-sat Framework

This section presents a performance optimized debugging framework using the discussed max-sat technique. The complexity of conventional debugging techniques such as SAT-based tools depend to a large extent on the number of suspects that must be considered. In the past, divide and conquer schemes based on the problem hierarchy have proven beneficial [3]. Here, the approximate max-sat approach can be used as a filter to remove the majority of the suspects by quickly finding over-approximate solutions. Subsequently, any exact debugging approach can be used and will benefit greatly by not having to consider all the original suspects during its analysis.

Any type of grouping can be used; however, in the remainder, clauses are grouped in sets of size  $G$  according to their corresponding circuit-level topology. Every group contains  $G$  clauses (except for one group that contains the remainder of the clauses in the CNF) from gates in close proximity to one another. For sequential circuits and multiple vectors, the group size is  $G \times$  [the total number of replications] as described in Sec. IV. Fig. 4 illustrates the flow of the proposed framework where the suspects are first filtered by the max-sat engine and then processed by the exact debugger. The optimal value of  $G$ , found experimentally, determines how the debugging effort is divided between the two stages.

## VI. EXPERIMENTS

The proposed framework is implemented in C++ using the max-sat algorithm (AllMCSes) in [8] and the SAT-based debugging engine in [5] as a second stage debugger. Six combinational and ten sequential circuits from ISCAS85, ISCAS89 and ITC99 benchmarks as well as OpenCores.org [9] are used

Circuit and debugging info					Debug	Max-sat20+debug						
name	# gates	# vecs	# repl.	# error locs	time (sec)	max-sat				debug time (sec)	total time (sec)	X improv.
						# grps	# suspects	% susp red	time (sec)			
mot-comb1	2,162	1	1	4	4.79	3	49	97.73%	0.03	0.05	<b>0.08</b>	59.88
mot-comb2	5,487	1	1	13	54.50	13	178	96.76%	0.13	0.24	<b>0.37</b>	147.30
mot-comb3	11,268	1	1	16	357.67	14	189	98.32%	0.27	0.47	<b>0.74</b>	483.34
c6288	3,466	1	1	75	67.96	48	536	84.54%	0.45	1.23	<b>1.68</b>	40.45
c7552	2,644	1	1	248	25.66	74	789	70.16%	0.11	3.11	<b>3.22</b>	7.97
c5315	1,884	1	1	11	4.83	7	99	94.75%	0.04	0.07	<b>0.11</b>	43.91
rsdecoder	12,041	1	2	11	572.68	7	126	98.95%	0.67	0.65	<b>1.32</b>	433.85
spi	2,012	1	21	19	80.54	12	194	90.36%	1.15	2.99	<b>4.14</b>	19.45
erp	2,449	1	3	13	36.09	11	179	92.69%	0.20	0.25	<b>0.45</b>	80.20
ac97	15,599	1	6	4	[TO]	3	58	99.63%	2.22	1.45	<b>3.67</b>	> 980.93
reactimer	265	1	512	7	<b>51.81</b>	6	89	66.42%	47.58	6.15	53.73	0.96
divider	5,248	1	15	4	1,160.39	3	52	99.01%	14.58	1.32	<b>15.90</b>	72.98
b14	5,695	1	22	45	1,377.86	36	627	88.99%	11.17	50.75	<b>61.92</b>	22.25
b15	8,938	1	13	32	[TO]	40	645	92.78%	96.99	65.82	<b>162.81</b>	> 22.11
s15850	10,481	1	2	19	747.36	12	183	98.25%	0.53	0.71	<b>1.24</b>	602.71
s38584	21,006	1	14	58	[TO]	34	566	97.31%	28.02	36.00	<b>64.02</b>	> 56.23
rsdecoder	12,041	4	8	11	[TO]	7	126	98.95%	2.88	2.01	<b>4.89</b>	> 736.20
spi	2,012	4	81	4	264.07	6	107	94.68%	4.95	4.39	<b>9.34</b>	28.27
erp	2,449	4	12	4	73.71	5	101	95.88%	0.82	0.52	<b>1.34</b>	55.01
ac97	15,599	4	23	4	[TO]	3	58	99.63%	9.95	5.05	<b>15.00</b>	> 240.00
reactimer	265	4	1,745	6	<b>172.30</b>	6	89	66.42%	2,845.80	21.48	2,867.28	0.06
divider	5,248	4	71	4	[TO]	3	52	99.01%	54.74	5.44	<b>60.18</b>	> 59.82
b14	10,114	4	1,216	-	[MO]	-	-	-	[MO]	-	-	-
b15	8,938	4	62	-	[TO]	-	-	-	[TO]	-	-	-
s15850	10,481	4	8	19	[TO]	12	183	98.25%	2.21	3.64	<b>5.85</b>	> 615.38
s38584	21,006	4	178	35	[MO]	20	365	98.26%	626.45	376.62	<b>1,003.07</b>	> 3.59

Fig. 5. Max-sat+debug versus standalone debugger

to construct several design debugging problems. The erroneous circuits are obtained by manually changing the functionality of a single gate at random. The failing test vectors are generated by running pseudo-random simulations until an erroneous response is observed. Experiments are conducted using both single and four failing test vectors. The performance of the proposed framework utilizing the max-sat pre-processing is compared against the efficiency of the SAT-based debugging engine in [5] without pre-processing. In all experiments, the size of the clause group error cardinality  $N_{cg}$  is set to one to find the single error sources. In addition to the groups created for the over-approximation, clauses are also grouped together based on the circuit replicas as discussed in Sec. IV. Experiments are conducted on a Pentium IV 2.8 GHz Linux platform with a 1GB memory limit and 3600 seconds time-out.

In order to determine the effectiveness of the overall debugging framework of Sec. V-A as a function of the group size  $G$ , experiments are conducted on several representative circuits. Fig. 6 (a) and (b) shows two such experiments, using circuit c6288 and mot-comb3, where three curves representing the run-times of the over-approximate max-sat stage, the exact debugging stage, and the combined run-times are presented for several group sizes. The run-time of max-sat increases abruptly as the group size becomes very small, and it reaches a maximum when the exact method is used (single-clause groups). However, as the group size increases, the run-time of the second stage debugger increases as it must consider many more suspects due to the over-approximation. The combined curve shows the total run-time of the overall framework is minimized with group sizes of roughly 10 to 20 clauses.

In the remaining, “max-sat20+debug” refers to the proposed

framework with a grouping size of  $G = 20$ . For sequential designs and multiple vectors the actual number of clauses per group is 20 times the number of circuit replicas. Figure 5 compares max-sat20+debug to the standalone debugger of [5]. Rows 1 – 6 report experiments with combinational circuits given a single failing test vector, and 7 – 16 (17 – 26) report experiments with sequential circuits given one (four) failing test vector(s). The first four columns contain the circuit’s name, its size in gates, the number of test vectors used, and the total number of circuit replicas needed. The fifth column ( $\# error locs$ ) gives the total number of potential error locations that could explain the faulty behavior of the circuit (the complete set). These are the locations expected to be returned by both approaches when available. The sixth column gives the run-time of the standalone debugger. An entry of [TO] denotes a time-out, and [MO] denotes a memory-out.

The remaining columns present the results of our proposed framework. The first four ( $\# grps$ ,  $\# suspects$ ,  $\% susp red$ , and  $time (sec)$ ) report the number of groups (of  $20 \times \# repl$ .

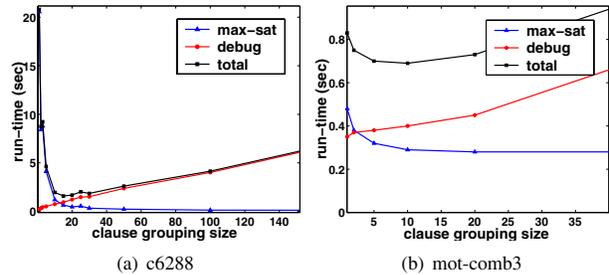


Fig. 6. Run-time versus clause grouping size

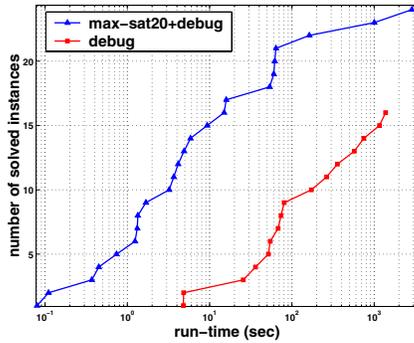


Fig. 7. max-sat20+debug versus debug

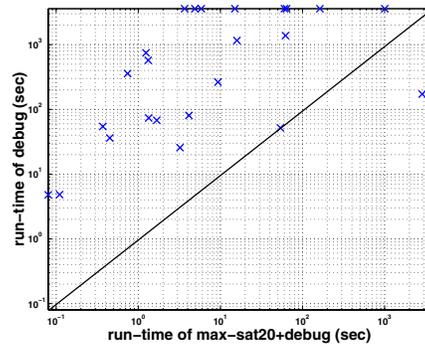


Fig. 8. max-sat20+debug versus debug

clauses) returned by the AllMCSes algorithm in any MCS; the number of *suspect* variables identified by those groups, each corresponding to a potential gate-level error source; the percent reduction in the number of suspect gates; and the run-time of this first stage. The true benefit of the proposed technique is evident when considering the number of suspects that are filtered by the first stage with relatively small run-time. For instance consider the circuit ac97 with a single vector. The approximation technique rules out 99.64% of the suspects in just 2.22 seconds. On average, the number of suspects is reduced by over 92%.

The run-time in seconds of the second stage debugger using the suspects of the first stage is shown in column *debug time (sec)*. Finally, the *total time (sec)* column shows the combined run-time of the proposed framework. This number is compared with the run-time of the standalone debugger in column six to get the improvements shown in the final column (*X improv.*).

These results demonstrate the overwhelming advantage of the proposed method over the standalone debugging engine as the run-times are reduced by an average of 200 times. For combinational circuits, the number of solved instances is increased from 16 to 24 out of 26, a 50% improvement, and for sequential circuits with one (four) test vector(s), the number of solved instances is increased from 7 (3) to 10 (8), a 43% (167%) improvement.

Fig. 7 plots the number of solved instances as a function of run-time on a logarithmic scale for max-sat20+debug and standalone debug. It can be seen that max-sat20+debug outperforms the standalone approach by roughly two orders of magnitude across all problems. Fig. 8 plots the total run-time of max-sat20+debug for each instance against the corresponding run-time of standalone debugger on a logarithmic scale. Clearly, most points lie above the 45° line which indicate the better performance of the proposed framework. Points on the upper border indicate the instances solved by max-sat20+debug but not solved by the standalone approach. The single point where the proposed framework fares essentially worse is caused by the large run-time of the first stage. Such cases can be addressed by increasing the group size  $G$ , thus reducing the difficulty for the AllMCSes algorithm.

## VII. CONCLUSION

This work presents an efficient two stage debugging framework which uses a novel max-sat problem formulation. First, it is shown that the debugging problem can be solved exactly with a max-sat formulation. The approach is extended for sequential circuits and for problems with multiple vectors. An over-approximation technique is developed to take advantage of the strengths of the max-sat techniques. This technique considers groups of clauses together and can thus make decisions based on the groups instead of the individual clauses. The over-approximation technique is used as a pre-processing step that filters the majority of suspects and reduces the problem complexity drastically for any debugger used in the second stage. Experiments demonstrate overwhelming run-time improvements of two orders of magnitude on average.

## REFERENCES

- [1] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.
- [2] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical path tracing - an alternative to fault simulation," in *DAC '83: Proceedings of the 20th conference on Design automation*, 1983, pp. 214–220.
- [3] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.
- [4] P. Bjesse and A. Boraly, "DAG-aware circuit compression for formal verification," in *Int'l Conf. on CAD*, 2004, pp. 42–49.
- [5] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir, "Debugging sequential circuits using Boolean satisfiability," in *Int'l Conf. on CAD*, 2004, pp. 204–209.
- [6] S.-Y. Huang, "A fading algorithm for sequential fault diagnosis," in *DFT '04: Proceedings of the Defect and Fault Tolerance in VLSI Systems, 19th IEEE International Symposium on (DFT'04)*, 2004, pp. 139–147.
- [7] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD*, vol. 21, no. 12, pp. 1377–1394, 2002.
- [8] M. Liffiton and K. A. Sakallah, "On Finding All Minimally Unsatisfiable Subformulas," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2005, pp. 32–43.
- [9] OpenCores.org, "http://www.opencores.org," 2006.
- [10] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.
- [11] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.

Seventh International Conference on Formal Methods in Computer-Aided Design  
**Industrial Strength SAT-based Alignability Algorithm  
for Hardware Equivalence Verification**

Daher Kaiss, Marcelo Skaba, Ziyad Hanna, Zurab Khasidashvili  
Intel, Israel Design Center, Haifa  
{dkaiss,smarcelo,zhanna,zurabk}@iil.intel.com

### Abstract

*Automatic synchronization (or reset) of sequential synchronous circuits is considered one of the most challenging tasks in the domain of formal sequential equivalence verification of hardware designs. Earlier attempts were based on Binary Decision Diagrams (BDDs) or classical reachability analysis, which by nature suffer from capacity limitations. A previous attempt to attack this problem using non-BDD based techniques was essentially a collection of heuristics aimed at toggling of the latches, and it is not guaranteed that a synchronization sequence will be computed if it exists.*

*In this paper we present a novel approach for computing reset sequences (and reset states) in order to perform sequential hardware equivalence verification between circuit models. This approach is based on the dual-rail modeling of circuits and utilizes efficient SAT-based engines for Bounded Model Checking (BMC). It is implemented in Intel's sequential verification tool, Seqver, and has been proven to be highly successful in proving the equivalence of complex industrial designs. The synchronization method described in this paper can be used in many other CAD applications, including formal property verification, automatic test generation, and power estimation.*

## 1. Introduction

In this work, we are concerned with *Formal Equivalence Verification (FEV)* of hardware designs. The goal is to verify "functional equivalence" between the specification (e.g., RTL) and the implementation (e.g., schematics) circuit models.

Most of the current generation FEV tools are limited by the requirement that the RTL be state matching with the schematics. That is, there must be a one-to-one correspondence between the state elements of the two compared designs. Therefore, a propositional tautology-checking procedure (e.g., a BDD [3] or SAT [12]-based solver), can be

employed to decide functional equivalence. This form of equivalence verification is known as *combinational equivalence* verification, as opposed to *sequential equivalence* where propositional decision procedures do not suffice and one needs other techniques to perform equivalence verification.

Combinational equivalence verification forces the designers to write detailed RTL in order to meet the state matching requirement. The low level of abstraction in the RTL model has a negative impact on the quality of the RTL [17]. Sequential equivalence verification comes to enable higher abstraction level of the RTL, and to reduce the effort of mapping of sequential elements during FEV.

While many algorithms and tools for several forms of sequential equivalence verification for circuits have existed for more than a decade [28, 2, 32, 36, 15, 19, 20], the industry has started to move towards full-scale sequential equivalence verification only recently [21, 26, 17, 22, 27]. The dominating concepts for sequential equivalence used in the industry all center around alignability equivalence [28]. Alignability equivalence verification can be performed in a compositional manner, where the equivalence of the compared circuits is derived from the equivalence of their corresponding sub-circuits [21, 22, 27]. Therefore, in this work we focus on alignability verification (or more precisely, on strong alignability [17] verification) of sub-circuits, possibly constrained with input properties, and do not discuss the compositionality related topics any further.

Both the BDD-based and SAT-based model checking algorithms require a set of initial states [10, 11, 37, 6, 7, 25, 1, 35]. However, in practice, an initial (or reset) state of a hardware design is not always available. In such cases, an automated technique for computing a reset state for hardware model checking would be an advantage. For alignability equivalence verification, both BDD-based and SAT-based methods have been proposed in the literature [28, 33, 21, 17, 27]. In some of the BDD-based methods [28, 27], computation of reset states of (sub-)circuits is not necessary, while in the SAT-based (strong) alignability verification method one first tries to synchronize the compared (sub-

circuits and then to compare the reset states for state equivalence [33, 21, 17]. The experimental data reported in the above works show that the SAT-based methods scale far better than the BDD-based methods; thus the importance of efficient SAT-based synchronization techniques is evident. Computation of initializing or reset sequences is a difficult problem [30, 33, 8, 29, 24, 18, 9]. Most of the previous algorithms are BDD-based. In [33], a SAT-based method for computing reset states is proposed. That method is essentially a set of heuristics aimed at toggling the latch control signals, as well as toggling some of the inputs and outputs. Toggling these signals is achieved by creating conditions (rather, invariant or safety properties) whose counter-examples can serve as input-vector sequences that guarantee that latch control signals will be asserted and input values will be propagated through the latches, and that some "important" inputs and outputs will toggle. As a result, there is a "good chance" that latches will be initialized. A Bounded Model Checking (BMC [1]) engine was used to generate the counter-examples, as for many sub-circuits the BDDs are not enough even to build the model-checking instances (the next-state functions and initial state relation).

The four-stage initialization algorithm in [33] is not sound in the presence of constraints on (sub-)circuit inputs and internal constraints, due to the unrestricted, random simulation at the first stage of initialization and "partial satisfaction of constraints" in the second stage. Moreover, this method is incomplete in that there is no guarantee (even theoretically) that an initial state or a reset state will be computed if one exists.

In this work we enhance the approach taken in [33] and present the first sound and complete SAT-based technique for computing a reset state. Unlike [33], we use the dual-rail modeling of circuits [4, 5], which allows us to more conveniently and precisely formulate the invariant properties whose counter-examples serve as (parts of the) synchronizing sequences that we are computing. We use the self-alignability ideas from [33] as a complementary method to 3-valued based initialization, and describe a hybrid automatic synchronization algorithm which is more efficient than the self-alignability technique alone and also succeeds in the cases where the 3-valued initialization alone fails.

This paper is organized as follows: In the next section we recall some basic definitions used in this work. In Section 3, we describe the dual-rail modeling of the circuits. Section 4 describes our algorithm for computing synchronizing sequences and performing strong alignability verification. Experimental results are discussed in Section 5. Conclusions appear in Section 6.

## 2. Preliminaries

A circuit design is modeled at the gate level in terms of combinational elements and storage elements. For simplicity, we assume that a storage element is a device that transfers its input to its output when a clock signal is high, and holds the output value when the clock signal is low. A *state*  $s$  of a circuit  $C$  is any one of the  $2^n$  assignments of boolean values to the  $n$  storage elements of  $C$ .

Without restricting generality, we assume that any circuit  $C$  has exactly one output,  $o$ , and a set of  $n$  storage elements  $L_1 \cdots L_n$ .  $C_1$  and  $C_2$  denote the specification and implementation circuits with outputs  $o_1$  and  $o_2$ , respectively.  $C_1$  and  $C_2$  have the same set of inputs (dummy inputs can be added, if necessary).  $C_{\Leftrightarrow}$  denotes the combined circuit of  $C_1$  and  $C_2$  – the *product machine* [13] with shared inputs and  $o = o_1 \Leftrightarrow o_2$  as the output.

We consider *ternary* modeling of circuit node values. The values can be one of the two *binary* values – T or F, or an *undefined* value –  $\perp$  (elsewhere also denoted by X). Given a binary input vector sequence  $\pi$ ,  $n(C, s, \pi)$  denotes the value of node  $n$  of a circuit  $C$  after 3-valued simulation of  $C$  with  $\pi$ , starting at state  $s$ . Similarly,  $C(s, \pi)$  denotes the (ternary) state into which  $\pi$  brings  $C$ , from state  $s$ . The *unknown state* of  $C$  is the state in which all the storage elements have the undefined value X. A *binary state* of a circuit  $C$  is a state in which all the state elements have binary values.

### 2.1. Alignability theory

Alignability equivalence [28] is a widely used concept of hardware equivalence verification. Its introduction was motivated by the fact that a power-up state of a hardware design cannot be predicted or controlled, therefore the design must be brought into a smaller set of states where the design is supposed to work correctly.

In this section we recall concepts related to alignability equivalence.

**Definition 2.1** *An initializing sequence of  $C$  is a sequence of binary inputs which, when applied to the unknown state  $X$  of  $C$ , brings  $C$  into a binary state.*

**Definition 2.2** *State  $(s_1, s_2)$  of the combined circuit  $C_{\Leftrightarrow}$  is an equivalent state (denoted by  $s_1 \simeq s_2$ ) if for any input sequence  $\pi$ ,  $o_1(C_1, s_1, \pi) = o_2(C_2, s_2, \pi)$ . States  $s_1$  and  $s_2$  are then called equivalent states of  $C_1$  and  $C_2$ .*

**Definition 2.3** ([28])

1. *A binary input sequence  $\pi$  is an aligning sequence for a combined state  $(s_1, s_2)$  of  $C_{\Leftrightarrow}$  if it brings  $C_{\Leftrightarrow}$  from state  $(s_1, s_2)$  into an equivalent state.*
2. *Circuits  $C_1$  and  $C_2$  are alignable, written  $C_1 \cong_{aln} C_2$ , if every state of  $C_{\Leftrightarrow}$  has an aligning sequence.*

It is shown in [28] that  $C_1 \cong_{aln} C_2$  iff there is a sequence, called a *universal aligning sequence*, that aligns any state of  $C_{\Leftrightarrow}$ . When the two circuits coincide  $C_1 = C_2 = C$ , then following [33] we will speak of the self-alignability of  $C$ . Note that  $C$  is self-alignable iff it is *weakly synchronizable*, as defined in [31]:

**Definition 2.4** A *weak synchronizing sequence* (ws-sequence for short) of a circuit  $C$  is an input vector sequence that brings  $C$  from any binary state to a subset of states  $S$ , called *ws-states* of  $C$ , such that all states of  $S$  are equivalent.

**Theorem 2.5** (Alignment Theorem [28]) Circuits  $C_1$  and  $C_2$  are alignable if and only if each circuit is weakly synchronizable and there is an equivalent pair  $s_1 \simeq s_2$  of states in  $C_1$  and  $C_2$ . The concatenation of ws-sequences of  $C_1$  and  $C_2$  is a ws-sequence for both of them and it weakly synchronizes  $C_1$  and  $C_2$  into equivalent ws-states (when  $C_1$  and  $C_2$  are alignable).

We now recall a more restrictive version of alignability, called *strong-alignability* [17].

**Definition 2.6** [23] A *reset or synchronizing sequence*  $\pi_r$  brings  $C$  from any binary state to a same state  $s_r$ , called a *reset or synchronizing state*.

Note that the set of ws-states and the set of synchronizing states are closed under state transition. Every synchronizing sequence is also a weakly synchronizing sequence. Therefore one can define strong-alignability as:

**Definition 2.7** [17] Circuits  $C_1$  and  $C_2$  are *strongly alignable* (denoted by  $C_1 \cong_{saln} C_2$ ) if each circuit is synchronizable and there is an equivalent pair  $(s_1, s_2)$  of states in  $C_1$  and  $C_2$ .

The concatenation of synchronizing sequences of  $C_1$  and  $C_2$  is a synchronizing sequence for both of them and it synchronizes  $C_1$  and  $C_2$  into equivalent states (when  $C_1$  and  $C_2$  are strongly alignable).

Strong alignability equivalence can be expressed in the following manner. Let circuit  $C_{\Leftrightarrow}$  with an output  $o$  be the product of two circuits  $C_1$  and  $C_2$ , and assume that  $\mathcal{S}$  is the set of states of  $C_{\Leftrightarrow}$ ; then:

$$C_1 \cong_{saln} C_2 \Leftrightarrow \exists \pi. \forall s \in \mathcal{S}. \left( \begin{array}{l} s_r = C_{\Leftrightarrow}(s, \pi) \wedge \\ \forall \gamma. o(C_{\Leftrightarrow}, s_r, \gamma) = \top \end{array} \right) \quad (1)$$

The strong alignability verification of two (sub-)circuits is done in two stages: (1) Compute a sequence which guarantees that when applied to a circuit  $C_{\Leftrightarrow}$ , it brings it to the same state. (2) Perform equivalence verification from the state(s) of the first stage. If the verification passes, then the models are declared (strongly) alignable.

The reasons for choosing strong alignability instead of the regular alignability for performing equivalence verification are driven by practical considerations. Assume that the second stage fails with a counter example. This failure can stem from two possible causes: (1) there is no sequence that aligns the two (sub-)circuits, or (2) this is a real counter example that needs to be debugged. Since it is not practical to require designers to debug counter examples which later on are root caused to a problematic initialization, we decided to work with strong alignability concept. For more methodological discussions concerning the usage of strong alignability, we refer the reader to [17].

### 3. Dual-rail modeling of hardware designs

In this section, we demonstrate on simple examples how Bounded Model checking (BMC) problems for hardware designs can be encoded using dual-rail modeling [4].

A circuit  $C$  is usually represented by a collection of *next-state functions* (NSFs) of the latches, by initial-state constraints on the latches, and by output functions. A NSF is a function of current and next-state values of inputs and latches. For example, consider the circuit  $C$  illustrated in Figure 1. It consists of four inputs  $a, b, c$  and  $clk$ , one latch,  $l$ , and the output of the circuit  $o$ . As usual, we denote the next-state version of a variable  $v$  by  $v'$ . So the function of the output  $o$  is  $l \vee c$ , while the NSF of the latch  $l$  is  $(clk' \wedge a' \wedge b') \vee (\neg clk' \wedge l)$ . Available convenient representations for next state functions can be BDDs or boolean expressions. The latter is a simple data structure for representing propositional logic formula. We adopted boolean expressions in our work since uniqueness of BDDs is not needed.

Modeling of sequential logic is done using a *compact* representation of infinite variable sequences. For a signal  $v$ , an infinite sequence of propositional variables  $\{v_0, v_1, v_2, \dots\}$  represents symbolically its sequential behavior. This allows one to reduce sequential verification problems to propositional satisfiability. The sequence representations can be unrolled to a desired depth  $k$ , producing  $k$  propositional variables  $\{v_0, v_1, \dots, v_{k-1}\}$ , which represent all the possible first  $k$  values of the signal  $v$ . This representation is suitable not only for modeling sequential behavior of inputs, but also for internal combinational signals, storage elements, and outputs. For example, for a given output  $o$  in Figure 1, the sequential behavior is represented by a disjunction of the sequences representing  $l$  and  $c$ . Similarly, we can define the behavior of any storage element by using NSFs.

Dual-rail modeling of circuits was introduced in [4, 5] to enable a more precise modeling of circuit operation, and to enable representation of all *ternary* values with BDDs via a binary encoding. Our adoption of dual rail modeling was

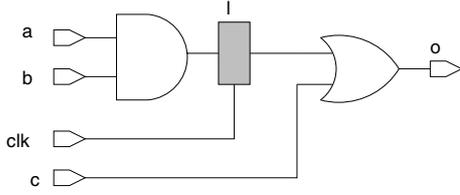


Figure 1: Example of latch and output functions

critical as our SAT solver is based on binary values  $(\mathbb{T}, \mathbb{F})$  while our algorithm needs ternary values  $(\mathbb{T}, \mathbb{F}, \perp)$ . We refer to [34, 16] for more information on dual-rail symbolic simulation.

Each ternary value  $v$  is encoded as a pair of binary values  $(v_h, v_l)$ , called the *high* and the *low* values. To avoid any confusion, we use  $\mathbb{F}^{dr}$ ,  $\mathbb{T}^{dr}$ , and  $\perp^{dr}$  to denote the dual-rail encoding of  $\mathbb{T}$ ,  $\mathbb{F}$  and  $\perp$ , respectively. And  $v^{dr} = (v_h, v_l)$  will denote the dual-rail encoding of a ternary variable  $v$ . The undefined value  $\perp^{dr}$  is encoded as a pair  $\perp^{dr} = (\mathbb{T}, \mathbb{T})$ . The truth constants are encoded by  $\mathbb{T}^{dr} = (\mathbb{T}, \mathbb{F})$  and  $\mathbb{F}^{dr} = (\mathbb{F}, \mathbb{T})$ . The pair  $\top^{dr} = (\mathbb{F}, \mathbb{F})$  encodes a *contaminated* or *over-specified* value.

In order to model sequential logic in dual-rail, it is enough to have dual-rail rules for the combinational operators and the *next state variable operator*, denoted by  $'$ . The dual-rail rules are as follows: Let  $x^{dr} = (x_h, x_l)$  and  $y^{dr} = (y_h, y_l)$  be dual-rail encoding of ternary variables  $x$  and  $y$ . Then

- $(x_h, x_l) \wedge (y_h, y_l) = (x_h \wedge y_h, x_l \vee y_l)$  ;
- $(x_h, x_l) \vee (y_h, y_l) = (x_h \vee y_h, x_l \wedge y_l)$  ;
- $\neg(x_h, x_l) = (x_l, x_h)$  ;
- $(x_h, x_l)' = (x'_h, x'_l)$ .

**Example 3.1** Let us compute  $x^{dr} \oplus \mathbb{T}^{dr}$  ( $\oplus$  stands for XOR)

$$\begin{aligned}
 x^{dr} \oplus \mathbb{T}^{dr} &= ((x_h, x_l) \wedge \neg(\mathbb{T}, \mathbb{F})) \vee (\neg(x_h, x_l) \wedge (\mathbb{T}, \mathbb{F})) \\
 &= ((x_h, x_l) \wedge (\mathbb{F}, \mathbb{T})) \vee ((x_l, x_h) \wedge (\mathbb{T}, \mathbb{F})) \\
 &= (x_h \wedge \mathbb{F}, x_l \vee \mathbb{T}) \vee (x_l \wedge \mathbb{T}, x_h \vee \mathbb{F}) \\
 &= (\mathbb{F}, \mathbb{T}) \vee (x_l, x_h) \\
 &= (\mathbb{F} \vee x_l, \mathbb{T} \wedge x_h) \\
 &= (x_l, x_h) \\
 &= \neg x^{dr}
 \end{aligned}$$

Below we omit the superscript  $^{dr}$  in dual-rail formulas.

To ensure that the inputs are always binary in a sequential instance, one needs to add, for any input variable  $i$ , a constraint  $i_h = \neg i_l$ . This guarantees that we do not introduce  $(\mathbb{F}, \mathbb{F})$  values in the instance. Further, if  $(\mathbb{F}, \mathbb{F})$  values are not introduced in constraints or in the properties, the NSF's cannot introduce them either (because the above

four operations cannot result in an  $(\mathbb{F}, \mathbb{F})$  value if the arguments are not over-constrained). Thus, for example, over-constrained values should not appear in a satisfying assignment found by a SAT-solver. An appearance of  $(\mathbb{F}, \mathbb{F})$  in a satisfying assignment indicates a bug, which is why we do not add a constraint to the instance forbidding over-constrained values on all variables.

A dual-rail NSF is a pair of NSF's of the high and low values. For example, the NSF of the latch in Figure 1 is represented in the dual rail encoding as follows:  $l'_h = (clk'_h \wedge a'_h \wedge b'_h) \vee (\neg clk'_h \wedge l_h) = (clk'_h \wedge a'_h \wedge b'_h) \vee (clk'_l \wedge l_h)$  and  $l'_l = (clk'_l \vee a'_l \vee b'_l) \wedge (clk'_h \vee l_l)$ .

## 4. Performing strong alignability verification

Given two circuits  $(C_1, C_2)$  (specification and implementation circuits), SAT-based strong alignability verification is performed in two stages: (1) compute the reset sequence for both circuits (which by itself produces the reset states) and (2) perform SAT-based state-equivalence verification from the computed reset states. In both stages we assume that the two circuits are modeled in one dual-rail model-checking instance as described in Section 3, and that the instance includes the *constraints* of the two circuits but not the property to be model-checked. The properties will be defined separately depending on the stages of the verification. Since the concatenation of reset sequences for  $C_1$  and for  $C_2$  is a reset sequence for both of them, for simplicity we will describe an algorithm for computing a reset state for one given circuit  $C$ .

### 4.1. The basic algorithm for computing a reset state

Assume a circuit  $C$  with exactly one output,  $o$ , with  $n$  inputs  $i_1, \dots, i_n$ , and  $m$  storage elements  $\mathcal{L} = \{L_1 \dots L_m\}$ . We will denote the value of a variable  $v$  (input, storage element or output) at time-frame  $k$  in the unrolled instance by  $v[k]$ . Computing a reset state consists of two stages: (a) initialization and (b) self-alignability.

#### (a) Initialization

In this stage, we try to find an initializing sequence  $\gamma_1$  for the circuit  $C$ . If this stage succeeds then there is no need to do the self-alignability stage, and we can proceed directly to the verification stage.

Recall the definition of an *initializing sequence* and the *unknown state* in Section 2. Every initializing sequence is a synchronizing sequence but the inverse is not true. This is true due to the monotonicity of the symbolic simulator, as any binary values resulting when simulating patterns containing X's would also result when the X's are replaced by any combination of 0's and 1's. [5]. An initializing sequence assumes the

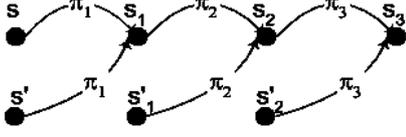


Figure 2: Performing self-alignability

unknown state as the starting state (all the storage elements are initialized with the X value). Therefore, for every storage element  $L$  modeled by  $(L_h, L_l)$  in the dual-rail modeling, we add to the model-checking instance initial-state constraints stating that  $L_h[0] = \top$  and  $L_l[0] = \top$ , which means that we are initializing the state element  $L$  with  $X = (\top, \top)$ . We are left with adding to the instance a property such that, if falsified at time-frame  $k$ , its counter-example satisfies the following formula:

$$\bigwedge_{L=(L_h, L_l) \in \mathcal{L}} (L_h[k] = \neg L_l[k]) \quad (2)$$

That is, if the counter-example is applied to the circuit  $C$  at the starting state  $X$ , all the latches will receive binary dual-rail values at the end of the simulation.

To check whether or not the property is falsifiable, one needs to choose a sufficiently large  $k$  (the diameter of  $C$ ) or use other SAT-based methods for full proofs, like the induction method [35] (since BDD-based model checking methods do not scale). In practice, however, SAT-based full-proof methods are computationally expensive compared to the regular BMC and we do not use them as a default. Instead, we chose a reasonably large bound  $k$  according to a heuristic and apply BMC till that bound. Furthermore, we modify the above property so that even if full initialization of  $C$  in  $k$  time steps is not possible, the BMC engine reports a counter-example that initializes a "large number" of latches. We then move to the self-alignability stage of our synchronization algorithm.

#### (b) Self-alignability

This stage attempts to complete the process of stage (a) by finding a synchronizing sequence  $\gamma_1\gamma_2$  for the circuit  $C$ . The computation is done iteratively: each new input sequence  $\pi_i$  is concatenated to the sequence already computed. We start from the state which was obtained after the initialization stage, so that part of the storage elements are not initialized yet. Assume that after applying the sequence  $\gamma_1$  of the initialization stage, out of the  $m$  storage elements in  $C$ , we have  $m'$  storage elements which are not initialized (have  $(\top, \top)$  value). Intuitively, the algorithm iteratively tries

to enumerate possible values for the  $m'$  uninitialized storage elements. The algorithm works as follows:

1. Set  $i = 1$ .
2. Replicate circuit  $C$  and produce new circuit  $C^{rep}$  in a way that the non-initialized  $m'$  storage element set in  $C$  (call it  $\mathcal{L}' \subseteq \mathcal{L}$ ) has a different corresponding encoding set in  $C^{rep}$  (call it  $\mathcal{L}'^{rep}$ ). That is, for every storage element  $L = (L_h, L_l)$  in  $C$ , produce a storage element  $L^{rep} = (L_h^{rep}, L_l^{rep})$  in  $C^{rep}$ . Our target now is to find a synchronizing sequence for the combined circuit built from  $C$  and its copy  $C^{rep}$ .
3. Let  $s_{init}$  be the ternary state of  $C$  at the end of the initialization stage (the latches in  $\mathcal{L}'$  have X values in  $s_{init}$ ). Assume that  $s$  and  $s'$  are states of  $C$  and  $C^{rep}$  such that the latches initialized during the initialization stage have the same concrete values as in  $s_{init}$ , and the remaining latches have binary dual-rail values. Furthermore, we add the following constraint to the model checking instance to state that  $s$  and  $s'$  are different states:

$$\bigvee_{L=(L_h, L_l) \in \mathcal{L}'} (L_h[0] = \neg L_h^{rep}[0]) \quad (3)$$

In addition, initial-state constraints defining the values of the initialized latches in  $s$  and  $s'$  are added to the instance.

4. Find a binary sequence  $\pi_i$  such that when applied to  $C$  and  $C^{rep}$  starting from  $s$  and  $s'$ , the result is the same binary state (See Figure 2). This is achieved by adding to the instance a property that, if falsifiable at time-frame  $k$ , the counter-example satisfies the following formula:

$$\bigwedge_{L=(L_h, L_l) \in \mathcal{L}} \left( \begin{array}{l} (L_h[k] = L_h^{rep}[k]) \wedge \\ (L_l[k] = L_l^{rep}[k]) \wedge \\ (L_h[k] = \neg L_l[k]) \end{array} \right) \quad (4)$$

5. If the property is not falsifiable till a bound  $k$ , then either the circuit is not synchronizable, or we did not choose a sufficiently large bound. We usually choose a large bound, so most probably the reason for the failure at this stage is due to the fact that the circuit is not synchronizable. The algorithm aborts with failure in this situation.
6. Otherwise, check whether there are two binary states of  $C$ ,  $C_{rep}$  such that the latches that are initialized after the initialization stages have the same values as in  $s_{init}$ , and when applying the sequence  $\pi_1 \dots \pi_i$  to both of them, the result is

two different binary states,  $s_i$  and  $s'_i$ . This is done by adding a property that, if falsifiable at bound  $k$ , the following formula is satisfied

$$\bigvee_{L=(L_h, L_l) \in \mathcal{L}} \left( \begin{array}{l} (L_h[k] = \neg L_h^{rep}[k]) \vee \\ (L_l[k] = \neg L_l^{rep}[k]) \end{array} \right) \quad (5)$$

and by restricting the behavior of the inputs of the circuit to comply with  $\pi_1 \cdots \pi_i$ .

7. If such states don't exist, then  $C$  is synchronizable and the sequence  $\gamma_1 \pi_1 \cdots \pi_i$  is a synchronizing sequence for it.
8. Otherwise, go to step 4 with  $s := s_i, s' := s'_i$  (s.t.  $s_i = C(\perp, \gamma_1 \pi_1 \cdots \pi_i)$  and  $s'_i = C_{rep}(\perp, \gamma_1 \pi_1 \cdots \pi_i)$ ) and with  $i := i + 1$ .

The algorithm is sound in the sense that if a sequence is found, it is guaranteed to be a synchronizing sequence. This is due to the iterative process which looks for new states that are not yet joined (i.e., transferred into the same state) by the already computed sequence. Furthermore, by the same argument, we can conclude that if in the last item of stage (b) of the algorithm we define  $s_i = C(s_{sym}, \gamma_1 \pi_1 \cdots \pi_i)$  and  $s'_i = C_{rep}(s_{sym}, \gamma_1 \pi_1 \cdots \pi_i)$  (rather than  $s_i = C(\perp, \gamma_1 \pi_1 \cdots \pi_i)$  and  $s'_i = C_{rep}(\perp, \gamma_1 \pi_1 \cdots \pi_i)$ ), where  $s_{sym}$  is a symbolic binary representation of a state in  $C$ , the algorithm becomes complete in the sense that if  $C$  is synchronizable, the algorithm will build a synchronizing sequence. When this algorithm was applied to real designs, the number of iterations was surprisingly small (most of the time 1 or 2 and always less than 10), which makes the algorithm very practical. This is partly due to the fact that many of the storage elements are already initialized during the initialization stage.

This hybrid system of combining initialization with self-alignability is indeed the breakthrough of our method, since initialization alone is insufficient and running self-alignability alone is not practical due to the large number iterations needed.

## 4.2. Performing strong alignability verification

Once the reset states of  $C_1$  and  $C_2$  are computed, we mainly use the induction-based algorithms [35] to check that the two reset states are state-equivalent. The induction based algorithms perform reasonably well when a full proof is sought. The instance should contain the NSFs, the initial-state constraints describing the reset states obtained during stage (1), the constraints, and a property stating that  $o_{\Leftrightarrow} = T$ .

Table 1: Single Outputs

Ckt	Inps	Latches	Result	#Self	Synch. (sec.)	Verify (sec.)
$o_1$	2283	40	EQUAL	2	300.5	617.1
$o_2$	396	884	EQUAL	0	2.2	40.9
$o_3$	111	133	EQUAL	0	0.3	10.5
$o_4$	969	112	EQUAL	0	2.4	9.6
$o_5$	96	132	EQUAL	1	1.3	8.7
$o_6$	277	873	EQUAL	0	1.6	2.9
$o_7$	381	402	EQUAL	2	0.1	0.8
$o_8$	398	984	EQUAL	0	2.4	0.2
$o_9$	275	748	EQUAL	0	0.8	0.1
$o_{10}$	23	72	Not Init.	2	0.9	0

Table 2: Total Run Times

Ckt	Gates	Latches	Outs	Inps	CPU(sec.)
$C_1$	362604	12693	389	397	4700
$C_2$	423070	23913	349	302	5722
$C_3$	157110	22080	572	578	2478

## 5. Results

The algorithms described are implemented in Intel's Sequential Equivalence Verifier, Seqver [17]. Seqver is being used to verify the next generation of Intel's processors, and has found many bugs. Tables 1 and 2 contain data about the size of the problems that were verified and the run time it took the tool to verify them. The run time was measured on a 2.4GHz Linux machine with 2GB memory. Table 1 illustrates the performance of the algorithm running on a single output. Note that for some of the outputs, initialization was not enough and we had to perform self-alignability (#self is the number of self-alignability iterations). The run time results show the run time for the synchronization and the verification stages. The number of iterations in the self-alignability stage is 1-2 on the average. Table 2 illustrates total run time for Seqver on representative circuits. To the best of our knowledge, no previous work has reported a tool that is able to perform (compositional) alignability verification on such large (sub-)circuits, and in such a short time, for circuits without predefined reset states.

## 6. Summary

We have given a description of a SAT-based algorithm for performing efficient alignability equivalence verification. This algorithm outperforms previous BDD based algorithms (which suffer from capacity limitations), and completes previous SAT-based attempts.

**Acknowledgments** We thank Gabriel Bischoff, Dinos Moundanos, Joonyoung Kim and Abhijit Jas for careful reading and valuable inputs.

## References

- [1] A. Biere, A. Cimatti, E. Clarke. *Symbolic model checking without BDDs*, Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1999.
- [2] G. P. Bischoff, K. S. Brace, S. Jain, R. Razdan. Formal implementation verification of the bus interface unit for the Alpha 21164 microprocessor, IEEE/ACM International Conference on Computer Design, ICCD 1997.
- [3] R.E. Bryant Graph-based algorithms for Boolean function manipulation, IEEE Trans.Computers, C-35(8), 1986.
- [4] R. E. Bryant. *Boolean Analysis of MOS Circuits*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-6, No. 4, 1987
- [5] R.E. Bryant, C.-J.H. Seger. *Digital circuit verification using partially-ordered state models*, Twenty-Fourth International Symposium on Multiple-Valued Logic, 1994.
- [6] G. Cabodi, P. Camurati, S. Quer. *Improved reachability analysis of large finite state machines*, ICCAD 1996.
- [7] G. Cabodi, P. Camurati. *Symbolic FSM traversals based on the transition relation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol 16, n. 5, 1997.
- [8] H. Cho, S.-W. Jeong, F. Somenzi, C. Pixley. *Synchronizing sequences and symbolic traversal techniques in test generation*, J. Electron. Test.: Theory & Applications, vol. 4, n. 2, 1993.
- [9] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda and G. Squillero. *A new approach for initialization sequences computation for synchronous sequential circuits*, IEEE VLSI in Computers and Processors, 1997.
- [10] O. Coudert, C. Berthet, J.C. Madre. *Verification of synchronous sequential machines based on symbolic execution*, Workshop of Automatic Verification Methods for Finite State Systems, 1989.
- [11] O. Coudert, J.C. Madre. *A Unified framework for the formal verification of sequential circuits.*, ICCAD 1990.
- [12] Davis,M., G.Logemann, D. Loveland, A machine program for theorem-proving, CACM 5(7), 1962.
- [13] G.D. Hachtel, F. Somenzi. *Logic Synthesis and Verification Algorithms*, Kluwer Academic Publishers, 1998.
- [14] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Reading, MA: Addison-Wesley, 1979.
- [15] S.-Y. Huang, K.-T. Cheng, K.-C. Chen. *Verifying sequential equivalence using ATPG techniques*, ACM Transactions on Design Automation of Electronic Systems, 2001.
- [16] R.B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*, Kluwer Academic Publishers, 2002.
- [17] D. Kaiss, S. Goldenberg, Z. Hanna, and Z. Khasidashvili. *Seqver: A Sequential Equivalence Verifier for Hardware Designs*, IEEE International Conference on Computer Design, ICCD 2006.
- [18] M. Keim, B. Becker and B. Stenner. *On the (non)-resetability of synchronous sequential circuits*, IEEE VLSI Test Symposium, 1996.
- [19] Z. Khasidashvili, Z. Hanna, TRANS: Efficient sequential verification of loop-free circuits, HLDVT 2002.
- [20] Z. Khasidashvili, and Z. Hanna. *SAT-Based methods for sequential hardware equivalence verification without synchronization*, BMC'03, ENTCS 89 (4), 2003.
- [21] Z. Khasidashvili, M. Skaba, D. Kaiss and Z. Hanna. *Theoretical Framework for Compositional Sequential Hardware Equivalence Verification in Presence of Design Constraints*, IEEE International Conference on Computer Aided Design, ICCAD 2004.
- [22] Z. Khasidashvili, M. Skaba, D. Kaiss and Z. Hanna. *Post-reboot equivalence and compositional verification of hardware*, FMCAD 2006.
- [23] Z. Kohavi. *Switching and Finite Automata Theory*, McGraw-Hill, 1978.
- [24] Y. Lu and I. Pomeranz. *Synchronization of large sequential circuits by partial reset*, IEEE VLSI Test Symposium, 1996.
- [25] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
- [26] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman. *Exploiting suspected redundancy without proving it*, Design Automation Conference, DAC 2005.
- [27] I-H. Moon, P. Bjesse, C. Pixley. A compositional approach to the combination of combinational and sequential equivalence checking of circuits without known reset states, DATE 2007.
- [28] C. Pixley. *A theory and implementation of sequential hardware equivalence*, IEEE transactions on CAD, 1992.
- [29] C. Pixley, and G. Beihl. *Calculating resettability and reset sequences*, International Conference on Computer-Aided Design, ICCAD 1991.
- [30] C. Pixley, S.-W. Jeong, G.D. Hachtel. *Exact calculation of synchronizing sequences based on binary decision diagrams*, IEEE transactions on Computer-Aided Design, vol. 13, 1994.
- [31] I. Pomeranz, S. M. Reddy. *On removing redundancies from synchronous sequential circuits with synchronizing sequences*, IEEE Trans. Comput., pp.20-32, 1996.
- [32] Ranjan R.K., V. Singhal, F. Somenzi, R.K. Brayton, Combinational verification for sequential circuits, DATE 1999.
- [33] A. Rosenmann and Z. Hanna. *Alignability equivalence of synchronous sequential circuits*, HLDVT, 2002.
- [34] C.-J. H. Seger, and R. E. Bryant. *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design, vol. 6, no. 2, 1995.
- [35] M. Sheeran, S. Singh, G. Stålmarck. *Checking safety properties using induction and a SAT-solver*, FMCAD, 2000.
- [36] V. Singhal, C. Pixley, A. Aziz, and R.K. Brayton, Theory of safe replacement for sequential circuits, IEEE Trans. on CAD of integrated circuits and systems, vol. 20, n.2, 2001.
- [37] H. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli. *Implicit enumeration of finite state machines using BDDs*, CAD'90, 1990.

# Boosting Verification by Automatic Tuning of Decision Procedures

Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu  
 Department of Computer Science, University of British Columbia  
 {hutter, babic, hoos, ajh}@cs.ubc.ca

**Abstract**—Parameterized heuristics abound in computer aided design and verification, and manual tuning of the respective parameters is difficult and time-consuming. Very recent results from the artificial intelligence (AI) community suggest that this tuning process can be automated, and that doing so can lead to significant performance improvements; furthermore, automated parameter optimization can provide valuable guidance during the development of heuristic algorithms. In this paper, we study how such an AI approach can improve a state-of-the-art SAT solver for large, real-world bounded model-checking and software verification instances. The resulting, automatically-derived parameter settings yielded runtimes on average 4.5 times faster on bounded model checking instances and 500 times faster on software verification problems than extensive hand-tuning of the decision procedure. Furthermore, the availability of automatic tuning influenced the design of the solver, and the automatically-derived parameter settings provided a deeper insight into the properties of problem instances.

**Index Terms**—Decision Procedures, Boolean Satisfiability, Search Parameter Optimization

## I. INTRODUCTION

The problems encountered in automated formal verification are typically hard. As with other computationally difficult problems, the key to practical solutions lies in the use of heuristic techniques. In the context of verification, decision procedures, which might be embodied as a BDD [1] package, a Boolean satisfiability (SAT) solver (e.g., [2]), or an automated theorem prover based on the Nelson-Oppen framework [3], all make use of various heuristics that have a crucial impact on their performance.

A high-performance decision procedure typically uses multiple heuristics that interact in complex ways. Some examples from the SAT-solving world include decision variable and phase selection, clause deletion, next watched literal selection, and initial variable ordering heuristics (e.g., [4], [5], [6]). The behavior and performance of these heuristics is typically controlled by parameters, and the complex effects and interactions between these parameters render their tuning extremely challenging.

During the typical development process of a heuristic solver, certain heuristic choices and parameter settings are tested incrementally, typically using a modest collection of benchmark instances that are of particular interest to the developer. Many choices and parameter settings thus made are “locked in” during early stages of the process, and typically, only few parameters are exposed to the users of the finished solver. In many cases, these users never change the default settings of

the exposed parameters or manually tune them in a manner similar to that used earlier by the developer.

Not surprisingly, this manual configuration and tuning approach typically fails to realize the full performance potential of a heuristic solver. In this paper, we present an alternative approach based on automated parameter optimization methods and demonstrate its benefits, which include substantial performance improvements, valuable guidance to the algorithm designer, and new insights into specific types of (SAT-encoded) verification problems.

Specifically, we explain how PARAMILS, a recent parameter optimization tool developed by Hutter et al. [7], was used during the development of SPEAR, a high-performance modular arithmetic decision procedure and SAT solver, which was developed in support of the CALYSTO static checker [8]. Although the performance of an early, manually-tuned version of SPEAR was comparable to that of a state-of-the-art SAT solver (MiniSAT 2.0 [9]), the use of PARAMILS ultimately lead to speedups between a factor of 4.5 and a factor of 500 due to the optimization of the search parameters. The use of PARAMILS also influenced the design of SPEAR and gave us some important insights about differences between (SAT-encoded) hardware and software verification problems; for example, we found that the software verification instances generated by the CALYSTO static checker required more aggressive use of SPEAR’s restart mechanism than the bounded model checking hardware verification benchmarks we studied.

While the results of our case study are interesting in their own right, it should be noted that our overall approach and the specific parameter optimization tool used in this study are very general and can be applied to any parameterized heuristic algorithm; the performance criterion that is automatically optimized can be runtime, precision, latency, or any other computable scalar metric.

## II. RELATED WORK

There are almost no publications on automated parameter optimization for decision procedures for formal verification. Seshia [10] explored using support vector machine (SVM) classification to choose between two encodings of difference logic into Boolean SAT. The learned classifier was able to choose the better encoding in most instances he tested, resulting in a hybrid encoding that mostly dominated the two pure encodings. The only other work we are aware of is unpublished, ad hoc work in industry.

There is, however, a fair amount of previous work on optimizing SAT solvers for particular applications. For example, Shtrichman [11] considered the influence of variable and phase decision heuristics (especially static ordering), restriction of the set of variables for case splitting, and symmetric replication of conflict clauses on solving bounded model checking (BMC) problems. He evaluated seven strategies on the Grasp SAT solver, and found that static ordering does perform fairly well, although no parameter combination was a clear winner. Later, Shacham and Zarpas [12] showed that Shtrichman’s conclusions do not apply to zChaff’s less greedy VSIDS heuristic on their set of benchmarks, claiming that Shtrichman’s conclusions were either benchmark- or engine-dependent. Shacham and Zarpas evaluated four different decision strategies on IBM BMC instances, and found that static ordering performs worse than VSIDS-based strategies. Lu et al. [13] exploited signal correlations to design a number of ATPG-specific techniques for SAT solving. Their technique showed roughly an order of magnitude improvement on a small set of ATPG benchmarks.

The automated parameter optimization tool used in our study has been recently introduced by Hutter et al. [7]; however, that work was more focused on theoretical properties of the algorithm and did not consider an application to a state-of-the-art solver for real-world problems. That work and the study presented here complement each other and also address two different communities. Very broadly, automated parameter optimization can be seen as a stochastic optimization problem that can be solved using a range of generic and specific methods [14], [15], [16]. However, these are either limited to algorithms with continuous parameters or algorithms with a small number of discrete parameters.

### III. ALGORITHM DEVELOPMENT AND MANUAL TUNING

The core of SPEAR is a DPLL-style [17] SAT solver, but with several novelties. For example, SPEAR features an elaborate clause prefetching mechanism that improves memory locality. To improve the prediction rate of the prefetching mechanism, Boolean constraint propagation (BCP) and conflict analysis have been redesigned to be more predictable. SPEAR also features novel heuristics for decision making, phase selection, clause deletion, and variable and clause elimination. In addition, SPEAR has several enhancements for software verification, such as support for modular arithmetic constraints [18], incrementality to enable structural abstraction/refinement [8], and a technique for identifying context-insensitive invariants to speed up solving multiple queries that share common structure [19]. Given all of these features, extensions, and heuristics, many components of SPEAR are parameterized, including the choice of heuristics, as well as enabling (or disabling) of various features: e.g., pure-literal rule, randomization, clause deletion, and literal sorting in freshly learned clauses. Thus, the optimization of these parameters is a challenging task.

After the first version of SPEAR was written and its correctness thoroughly tested, its developer, Domagoj Babić, spent one week on manual performance optimization, which

involved: (i) optimization of the implementation, resulting in a speedup by roughly a constant factor, with no effects on the search parameters, and (ii) manual optimization of roughly twenty search parameters, most of which were hard-coded and scattered around the code at the time.

The manual parameter optimization was a slow and tedious process done in the following manner: the SPEAR developer collected several medium-sized benchmark instances which it could solve in at most 1000 seconds and attempted to come up with a parameter configuration that would result in a minimum total runtime on this set. The benchmark set was very limited and included several medium-sized BMC and some small software verification (SWV) instances generated by the CALYSTO static checker [8].<sup>1</sup> Such a small set of test instances facilitates fast development cycles and experimentation, but has many disadvantages.

Quickly it became clear that implementation optimization gave more consistent speedups than parameter optimization. Even on such a small set of benchmarks, the variations due to different parameter settings were huge. We even found one case (Alloy analyzer [20] instance handshake.als.3) where the difference of floating point rounding errors between Intel’s non-standard 80-bit and IEEE 64-bit precision resulted in an extremely large difference in the runtimes on the same processor. The same instance was solved in 0.34 sec with 80-bit precision and timed out after 6000 sec with 64-bit precision. The difference in rounding initially caused minor differences in variable activities, which are used to compute the dynamic decision ordering. Those minor differences quickly diverged, pushing the solver into two completely different parts of search space. Since most parameters influence the decision heuristics in some way, the solver might be equally sensitive to parameter changes.<sup>2</sup>

Given the costly and tedious nature of the process, no further manual parameter optimization was performed after finding a configuration that seemed to work well on the chosen test set.

To assess the performance of this manually tuned version of SPEAR, we ran it against MiniSAT 2.0 [9], the winner of the industrial category of the 2005 SAT Competition and of the 2006 SAT Race. In this experiment, we used two instance sets introduced in detail later in Sec. V: bounded model checking (BMC) and software verification (SWV). As can be seen from the runtime correlation plots shown in Figure 1, both solvers perform quite similarly for bounded model checking and easy software verification instances. For difficult software verification instances, however, MiniSAT clearly performs better. This seems to be the effect of focusing the manual tuning on a small number of easy instances.

For most decision procedures, the process of finding default (or hard-coded) parameter settings resembles the manual tuning described above. Furthermore, most users of these tools

<sup>1</sup>Small instances were selected because CALYSTO tends to occasionally generate very hard instances that would not be solved within a reasonable amount of time.

<sup>2</sup>This emphasizes the need to find parameter settings that lead to more robust performance, with different random seeds, as well as across instances.

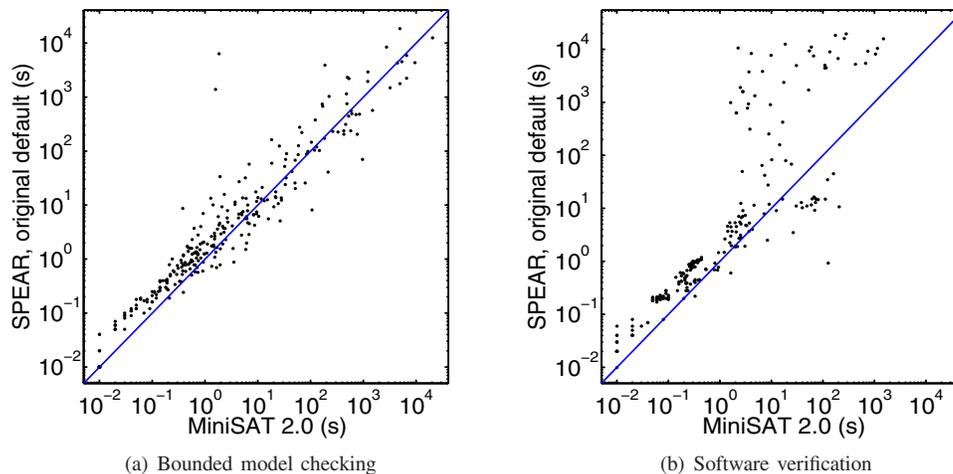


Fig. 1. MiniSAT 2.0 vs. SPEAR using its original, manually tuned default parameter settings. (a) The two solvers perform comparably on bounded model checking instances, with average runtimes of 298 seconds (MiniSAT) vs. 341 seconds (SPEAR) for the instances solved by both algorithms. (b) Performance on easy and medium software verification instances is comparable, but MiniSAT scales better for harder instances. The average runtimes for instances solved by both algorithms are 30 seconds (MiniSAT) and 787 seconds (SPEAR).

do not change these settings, and when they do, they typically apply the same manual approach.

#### IV. PARAMETER OPTIMIZATION BY LOCAL SEARCH

The tool we chose to use for automatically optimizing parameter settings in SPEAR has recently been developed in the Artificial Intelligence community [7]; in the following, we briefly introduce the underlying PARAMILS algorithm (further details and some theoretical background can be found in the paper by Hutter et al. [7]).

PARAMILS is motivated by the following manual parameter tuning technique often used by algorithm developers:

- Start with some parameter configuration
- Iteratively, modify one algorithm parameter at a time, keeping the modification if performance on a given benchmark set improves and undoing it otherwise.
- Terminate when no single parameter modification yields an improvement, or when the best configuration found so far is considered “good enough”.

Notice that this is essentially a simple hill-climbing local search process, and as such it will typically terminate in a locally, but not globally optimal parameter configuration, in which changing any single parameter value will not achieve any performance improvement. However, since parameters of heuristic algorithms are typically not independent, changing two or more parameter values at the same time may still improve performance.

The problem of local optima is ubiquitous in local search, and many approaches have been developed to effectively deal with them; one of these approaches is *Iterated Local Search (ILS)* [21], [22], which provides the basis for PARAMILS. ILS essentially alternates a subsidiary local search procedure (such as simple hill-climbing) with a perturbation phase, which lets the search escape from a local

minimum. Additionally, an acceptance criterion is used to decide whether to continue the search from the most recently discovered local minimum or from some earlier local minimum. More precisely, starting from some initial parameter configuration, PARAMILS first performs simple hill-climbing search until a local minimum  $c$  is reached, and then it cycles through the following phases:

- 1) apply perturbation (in the form of multiple random parameter changes);
- 2) perform simple hill-climbing search until a new local minimum  $c'$  is reached;
- 3) accept the better of the two configurations  $c$  and  $c'$  as the starting point of the next cycle.

PARAMILS thus performs a biased random walk over locally optimal parameter configurations. To determine the better of two configurations, it can use arbitrary scalar performance metrics, including expected runtime, expected solution quality (for optimization algorithms), or any other statistic on the performance of the algorithm to be tuned when applied to instances from a given benchmark set. This benchmark set is called the *training set*, in contrast to the *test sets* we used later for evaluating the final parameter configurations obtained from PARAMILS (as is customary in the empirical evaluation of machine learning algorithms, training and test sets are strictly disjoint).

Clearly, the choice of the training set has important consequences for the performance of PARAMILS. Ideally, a homogenous training set would be chosen, i.e., one in which the impact of parameter settings on the performance of the algorithm to be tuned (here, SPEAR) is similar for all instances in the set. In that case, it would be sufficient and ‘safe’ to evaluate and compare parameter configurations by running the solver on a small number of instances. In practice, however, ‘interesting’ instance sets may not be homogenous,

and therefore larger training sets may be required to achieve a reasonably unbiased evaluation of parameter configurations.

BASICILS( $N$ ) is a simple version of PARAMILS that uses a training set of  $N$  instances, where the choice of  $N$  has a major impact on the efficacy of the tuning process. For small  $N$ , there is a risk of over-fitting, i.e., good parameter configurations determined for the corresponding small sets may be overly specific to the training set and not work well for any other problem instances. For large  $N$ , however, the evaluation of each parameter configuration becomes costly, which can severely limit the number of search steps that can be practically performed by PARAMILS (and hence reduce the quality of the final parameter configuration returned by the tuning algorithm).

FOCUSEDILS is a more advanced version of PARAMILS. It adaptively chooses the number of training instances to use for each parameter setting: while poor settings can be discarded after a few algorithm runs, promising ones are evaluated on more instances. This mechanism avoids over-fitting to the instances in the training set. (For details, see [7].) In tuning SPEAR, we initially used BASICILS(300) and later employed the more advanced FOCUSEDILS.

## V. AUTOMATED PARAMETER OPTIMIZATION

We performed two sets of experiments: automated tuning of SPEAR on a general set of instances for the 2007 SAT competition and application-specific tuning for two real-world benchmark sets.

### A. Benchmark Sets and Experimental Setup

We employed two sets of problems of immense practical importance: hardware bounded model checking and software verification. Specifically, our set of BMC instances consists of 754 IBM BMC instances created by Zarpas [23], and our SVW benchmark set is comprised of 604 verification conditions generated by the CALYSTO static checker [8].

Both instance sets, BMC and SVW, were split 50:50 into disjoint training and test sets. Only the training sets were used for tuning, and all results in this paper are for the test sets. All reported experiments were carried out on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1. Reported times are CPU times per single CPU. Runs are terminated after 10 CPU hours or when they run out of memory and start swapping; we count both of these conditions as time-outs.

### B. Search Parameters

The availability of automatic parameter tuning encouraged us to parameterize many aspects of SPEAR. The first automatically tuned version exposed only a few important parameters, such as restart frequencies and variable priority increments. The results of automated tuning of those first versions of SPEAR prompted its developer to expose more and more search parameters, up to the point where not only every single hard-coded parameter was exposed, but also a number of new parameter-dependent features were incorporated. This process

not only significantly improved SPEAR’s performance, but also has driven the development of SPEAR itself.

The resulting version of SPEAR used for the experiments reported in the following has 26 parameters:

- 7 types of heuristics (with the number of different heuristics available shown in parentheses):
  - Variable decision heuristics (20)
  - Heuristics for sorting learned clauses (20)
  - Heuristics for sorting original clauses (20)
  - Resolution ordering heuristics (20)
  - Phase selection heuristics (7)
  - Clause deletion heuristics (3)
  - Resolution heuristics (3)
- 12 double-precision floating point parameters, including variable and clause decay, restart increment, variable and clause activity increment, percentage of random variable and phase decisions, heating/cooling factors for the percentage of random choices, etc.
- 4 integer parameters which mostly control restarts and variable/clause elimination.
- 3 Boolean parameters which enable/disable simple optimizations such as the pure literal rule.

For each of SPEAR’s floating point and integer parameters we chose lower and upper bounds on reasonable values and considered a number of values spread uniformly across the respective interval. This number ranges from three to eight, depending on our intuition about importance of the parameter. The total number of possible combinations after this discretization is  $3.78 \times 10^{18}$ . By exploiting some dependencies between parameters, we reduced the number of configurations that we consider in this paper to  $8.34 \times 10^{17}$ .

### C. SAT Competition Tuning

The first round of automatic parameter optimization was done in the context of preparing a version of SPEAR for submission to the 2007 SAT Competition. The first two authors used this as a case study in parameter optimization for real-world problem domains: the SPEAR developer provided an executable of SPEAR and information about its parameters as well as approximate ranges of reasonable values for each of them; the default parameter configuration, however, was not revealed. The goal of this study was to see whether the performance achieved with automatic methods could rival the performance achieved by the manually engineered default parameters.

Since the optimization objective was to achieve good performance on the industrial benchmarks of the 2007 SAT Competition (which were not disclosed before the solver submission deadline), we used a collection of instances from previous competitions for tuning: 176 industrial instances from the 2005 SAT Competition, 200 instances from the 2006 SAT Race, as well as 30 SVW instances generated by the CALYSTO static checker. A subset of 300 randomly selected instances was used for training, and the remaining 106 test instances provided an unbiased performance estimate of SPEAR’s performance with

the tuned parameter configuration. Since the SAT competition rules reward per-instance performance relative to other solvers, the optimization objective used in this phase was geometric mean speedup over SPEAR with the (manually optimized) default parameter settings.

We ran a single run of BASICILS(300) for three days on the 300 designated training instances, and used the parameter configuration with the best training set performance found within that time; we refer to this parameter configuration as `SatComp`. During tuning, we took the risk of setting a low cutoff time of 10 seconds for each single algorithm run in order to save time. This bore the possibility of over-tuning the solver for good performance on short runs but poor performance on longer runs, and we expected that parameter configuration `SatComp` may be too aggressive and might perform poorly on harder instances.

However, our experimental results indicate that the opposite is the case, namely that SPEAR’s performance scales better with the `SatComp` parameter settings than with the default settings. The fact that these results contradicted the intuition of the algorithm’s developer illustrates clearly the limitations of even an expert’s ability to comprehend the complex interplay between the many parameters of a sophisticated heuristic algorithm such as SPEAR.

On the 106 test instances used to assess the result of our SAT competition tuning, `SatComp` achieved a geometric mean speedup of 21% over SPEAR’s default parameter settings and showed much better scaling with instance hardness. Figure 2 demonstrates that this speedup carries over to both our verification benchmark sets: `SatComp` performs better than the SPEAR default on BMC (with an average speedup factor of about two) and clearly dominates it for SWV (with an average speedup factor of about 78).

#### D. Application-specific Tuning

While general tuning on a mixed set of instances as performed for the 2007 SAT Competition resulted in a solver with strong overall performance, in practice, one often mostly cares about excellent performance on a specific type of instances, such as BMC or SWV. For this reason we performed a second set of experiments — tuning SPEAR for these two specific sets of problems. Since users typically care most about an algorithm’s total runtime, we used average (arithmetic mean) runtime as our optimization objective in this tuning phase.

For both sets, during training we chose a cutoff of 300 seconds, which according to SPEAR’s internal book-keeping mechanisms turned out to be sufficient for exercising all techniques implemented in the solver. In order to speed up the optimization, in the case of BMC we removed 95 hard instances from the training set that could not be solved by SPEAR with its default parameter configuration within one hour, leaving 287 instances for training.

We performed parameter optimization by running 10 parallel copies of FOCUSEDILS on a cluster, for three days in the case of SWV and for two days for BMC. For each instance set,

we picked the parameter configuration with the best training performance after that time.

Figure 3 demonstrates that these application-specific parameter configurations perform even better than the optimized settings for the SAT competition, `SatComp`. SPEAR’s performance is boosted for both application domains, by an average factor of over 2 for BMC and over 20 for SWV; the scaling behavior also clearly improves, especially for SWV.

Figure 4 shows the total effect of automatic tuning by comparing the performance of SPEAR with the (manually optimized) default settings against that achieved when using the parameter configurations tuned parameters for the BMC and SWV benchmark sets. For both sets, the scaling behavior of the tuned version is much better and on average, large speedups are achieved — by a factor of 4.5 for BMC and 500 for SWC. SPEAR with the default settings even times out on four SWV instances after 10 000 seconds, while the tuned version solves every single instance in less than 20 seconds.

Figure 5 summarizes the performance of MiniSAT 2.0 (which we used as a baseline) and SPEAR with parameter settings default, `SatComp`, and specifically tuned for BMC and SWV. Notice that the versions of SPEAR specifically tuned for BMC and SWV also clearly outperform MiniSAT: for BMC, SPEAR solves two additional instances and is faster by a factor of three on average; for SWV, the speedup factor is over 100. For both benchmark sets, scatter plots (not shown here) also reveal much better scaling behavior of the specifically tuned versions of SPEAR.

## VI. DISCUSSION

Automated parameter tuning provided us with new insights into properties of the benchmark instances used in our study and influenced the design of SPEAR. These insights arise from considering characteristic differences between the optimized parameter configurations for the BMC and SVW instances.

Although we have limited knowledge about the high-level features of the IBM BMC instances, we made some interesting observations. The best decision heuristic that we found for these instances picks variables with higher activity, and ties are resolved by choosing the one with a smaller product of positive and negative occurrences. We also found that the IBM BMC instances favor less aggressive restarts than the SVW instances, implying that the decision heuristic tends to find better variable orderings. The best phase selection heuristic we found for BMC instances aggressively picks the phase so as to minimize the number of watched clauses that need to be traversed in order to find the next watched literal. This heuristic minimizes the number of clauses that BCP needs to analyze, and its effectiveness on this hard set of instances did not surprise us. Finally, we observed that a small amount of randomness helps performance — roughly 5% of phase and variable decisions were done randomly before the first restart. The most effective strategy scales down the percentage of random decisions by a factor of 0.7 at each restart (which resembles the idea of simulated annealing).

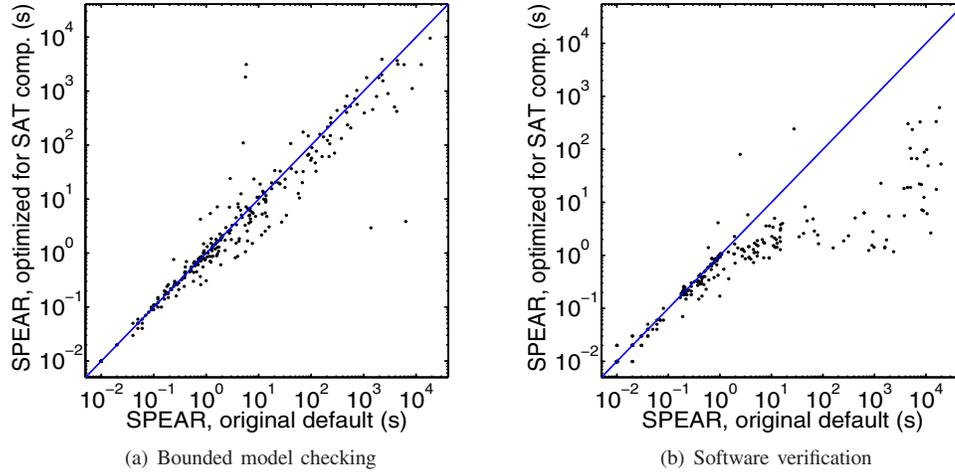


Fig. 2. Improvements by automated parameter optimization on a mix of industrial instances: SPEAR with the original default parameter configuration vs. SPEAR with configuration `Satcomp`. (a) Even though a few instances can be solved faster with the SPEAR default, parameter configuration `Satcomp` is considerably faster on average (mean runtime 341 vs. 223 seconds). Note that speedups are larger than they may appear in the log-log plot: for the bulk of the instances `Satcomp` is about twice as fast. (b) `Satcomp` improves much on the scaling behavior of the SPEAR default, which fails to solve four instances in 10000 seconds. Mean runtimes on the remaining instances are 787 seconds vs. 10 seconds, a speedup factor of 78.

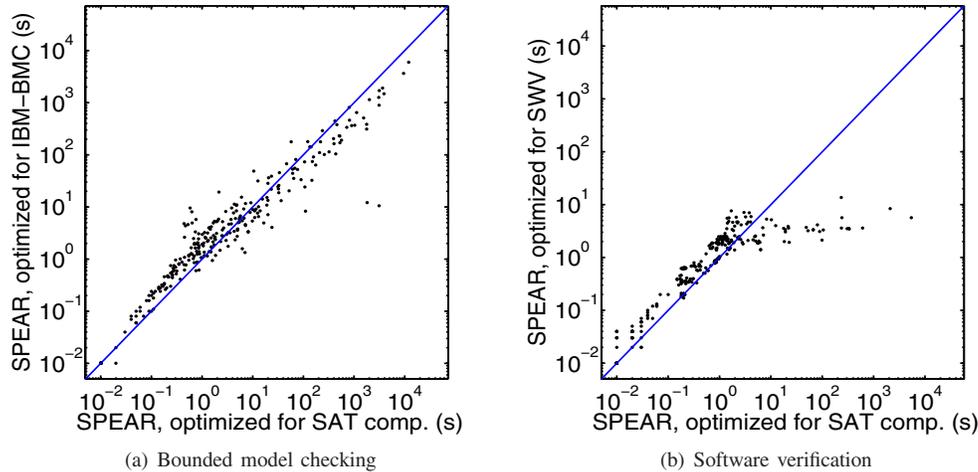


Fig. 3. Improvements by automated parameter optimization on specific instance distributions: SPEAR with configuration `Satcomp` vs. SPEAR with parameters optimized for the specific applications BMC and SWV. Results are on independent test sets disjoint from the instances used for parameter optimization. (a) The parameter configuration tuned for set BMC solved four instances for which configuration `Satcomp` timed out after 10000 seconds. For the remaining instances, mean runtimes are 223 seconds (`Satcomp`) and 96 seconds (specific tuning for BMC), a speedup by more than a factor of two. (b) Both parameter settings solved all 302 instances, mean runtimes are 36 seconds (`Satcomp`) and 1.5 seconds (tuned for SWV), a speedup factor of 24.

Since we are intimately familiar with the CALYSTO static checker, we are able to provide a deeper analysis for the software verification instances. CALYSTO performs aggressive common subexpression elimination, virtually eliminating all symmetries. It also propagates all constants. CALYSTO queries correspond to path- and context-sensitive verification conditions, which have deep and rich Boolean structure, with many expensive operations (like division and multiplication) sprinkled around. The queries can be represented at a high level as single-rooted acyclic graphs. Experimental results (see [8]) suggest that the probability of infeasibility of a single path

starting from the root of the formula is proportional to the length of the path — the longer the path, the more likely it is that it is infeasible. This can be exploited by a SAT solver by focusing the search on the expressions that are closer to the root of the tree.

SWV instances prefer an activity-based heuristic that resolves ties by picking the variable with a larger product of occurrences. This heuristic might seem too aggressive, but helps the solver to focus on the most frequently used common subexpressions. It seems that a relatively small number of expressions play a crucial role in (dis)proving each verifica-

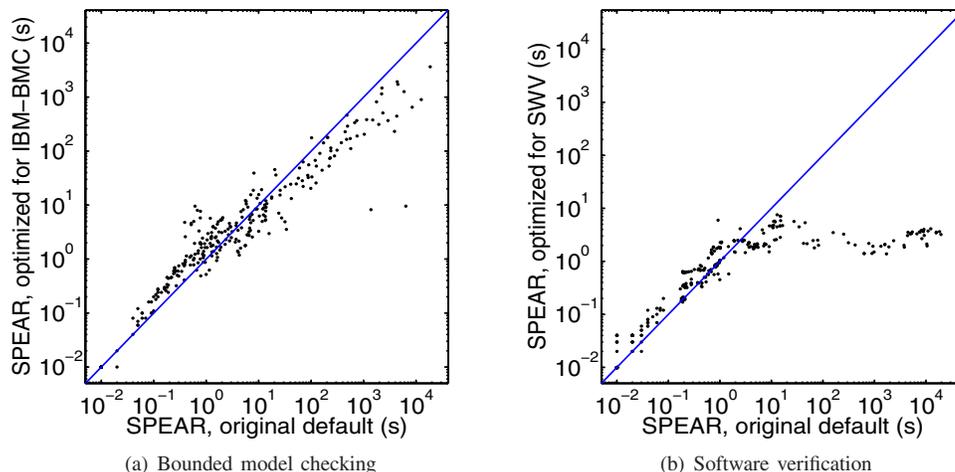


Fig. 4. Overall improvements achieved by automatic tuning: SPEAR with its manually engineered default parameter configuration vs. the optimized versions for sets BMC and SWV. Results are on test sets disjoint from the instances used for parameter optimization. (a) The default timed out on 90 instances after 10 000 seconds, while the tuned configuration solved four additional instances. For the instances that the default solved, mean runtimes are 341 seconds (default) and 75 seconds (tuned), a speedup factor of 4.5. (b) The default timed out on four instances after 10 000 seconds, the tuned configuration solved all instances in less than 20 seconds. For the instances that the default solved, mean runtimes are 787 seconds (default) and 1.35 seconds (tuned), a speedup factor of over 500.

Solver	Bounded model checking		Software verification	
	#(solved)	runtime for solved	#(solved)	runtime for solved
MiniSAT 2.0	289/377	360.9	302/302	161.3
SPEAR original	287/377	340.8	298/302	787.1
SPEAR general tuned	287/377	223.4	302/302	35.9
SPEAR specific tuned	291/377	113.7	302/302	1.5

Fig. 5. Summary of Results. For each solver and instance set, #(solved) denotes the number of instances solved within a CPU time of 10 hours, and the runtimes are the arithmetic mean runtimes for the instances solved by that solver. (Geometric means were not meaningful here, as all solvers solved a number of easy instances in “0 seconds”; arithmetic means better reflect practical user experience as well.) If an algorithm solves more instances, the shown average runtimes include more, and typically harder, instances. Note that the averages in this table differ from the runtimes given in the captions of Figures 1-4, because averages are taken with respect to different instance sets: for each solver, this table takes averages over all instances solved by that solver, whereas the figure captions state averages over the instances solved by both solvers compared in the respective figure.

tion condition, and this heuristic quickly narrows the search down to such expressions. The SWV instances favored very aggressive restarts (first after only 50 conflicts), which in combination with our experimental results shows that most such instances can be solved quickly if the right order of variables is found. A simple phase selection heuristic (always assign FALSE first) seems to work well for SWV, and also produces more natural bug traces (small values of variables in the satisfying assignments). The SWV instances correspond to NULL-pointer dereferencing checks, and this phase selection heuristic attempts to propagate NULL values first (all FALSE), which explains its effectiveness. SWV instances prefer no randomness at all, which is probably the result of joint development of CALYSTO and SPEAR as a highly optimized tool chain for software verification.

The use of automated parameter optimization also influenced the design of SPEAR in various ways. An early version of SPEAR featured a nascent implementation of clause and variable elimination. Prior to using automated tuning, these mechanisms did not consistently improve performance,

and therefore, considering the complexity of finalizing their implementation, the SPEAR developer considered removing them. However, these elimination techniques turned out to be effective after parameter tuning found good heuristic settings to regulate the elimination process. Another feature that was considered for removal was the pure literal rule, which ended up being useful for BMC instances (but not for SWV). Similarly, manual optimization gave inconclusive results about randomness, but automated optimization found that a small amount of randomness actually does help SPEAR in solving BMC (but not SWV) instances.

## VII. CONCLUSIONS

In this work, we have demonstrated that by using a general parameter optimization method, PARAMILS, which is based on the idea of iterated local search in parameter configuration space, major performance improvements of a high-performance SAT solver, SPEAR, can be achieved. We believe that the resulting optimized version of SPEAR represents a considerable improvement in the state of the art of solving

decision problems from hardware and software verification using SAT-solvers. Tuning SPEAR on a general set of industrial instances from previous SAT competitions already resulted in large improvements when compared to SPEAR's manually optimized default parameter setting. The greatest improvements, however, were achieved when tuning was performed on a specific, relatively homogenous class of problem instances. Average runtimes were reduced by a factor of 4.5 for bounded model checking instances and a factor of over 500 for software verification instances (see Figure 4). It is worth noting that prior to applying our automated tuning approach, considerable time had been invested by its author to manually tune SPEAR. This indicates that automated parameter optimization can be considerably more effective than manual tuning, and that the use of automated tuning procedures such as PARAMILS not only frees the algorithm designer (and user) from the typically tedious and time-consuming manual tuning task, but also helps to better exploit the full performance potential of a highly parameterized heuristic solver.

Not too surprisingly, our experimental results suggest that optimized search parameters are benchmark-dependent — which highlights the advantages of automated parameter tuning over the conventional manual approach. Furthermore, parameter tuning is obviously engine-dependent, due to complex interactions between various mechanisms implemented in a typical decision procedure.

We also illustrated how the use of automated parameter optimization provided guidance in the development of SPEAR and in particular encouraged its developer to expose a large number of parameters that could then be optimized. We are convinced that similar benefits will arise when applying our general approach in the development of other heuristic algorithms. Finally, comparing specifically optimized parameter configurations, we gained some insights into which components of SPEAR were particularly effective on the hardware and software verification instances considered here.

In future work, we intend to further explore the role of local search and machine learning strategies that support algorithm design and engineering tasks. We believe that the tuning procedure can be further improved, for example, by combining ideas from our current local-search-based approach with concepts from racing procedures, or by incorporating techniques from experimental design. We also see significant potential in instance-specific tuning methods, which use machine learning techniques to find good parameter settings for a given problem instance [24], and in reactive tuning strategies, which adapt parameter settings while a solver is running (utilizing information gathered while trying to solve the given instance) [25]. Finally, considering that many other design and engineering tasks involve heuristic algorithms, we are convinced that the use of automated algorithm configuration and parameter optimization procedures can lead to similarly substantial performance improvements as demonstrated here and hope to collect further evidence for this claim in the near future.

## REFERENCES

- [1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.
- [2] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification." *STTT: International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, April 2005.
- [3] G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1979.
- [4] J. P. M. Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," in *EPIA '99: Proc. of the 9th Portuguese Conference on Artificial Intelligence*, ser. LNCS, vol. 1695. Springer, 1999, pp. 62–74.
- [5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *DAC '01: Proc. of the 38th conference on Design automation*. ACM Press, 2001, pp. 530–535.
- [6] A. Bhalla, I. Lynce, J. de Sousa, and J. Marques-Silva, "Heuristic backtracking algorithms for SAT," in *MTV '03: Proc. of the 4th International Workshop on Microprocessor Test and Verification: Common Challenges and Solutions*, 2003, pp. 69–74.
- [7] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *AAAI '07: Proc. of the Twenty-Second Conference on Artificial Intelligence*, 2007, pp. 1152–1157.
- [8] D. Babić and A. J. Hu, "Structural Abstraction of Software Verification Conditions," in *Computer Aided Verification: 19th International Conference, CAV 2007*, ser. LNCS, vol. 4590. Springer, 2007, pp. 366–378.
- [9] N. Eén and N. Sörensson, "An extensible SAT solver," in *SAT '03: Proc. of the 6th International Conference on theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 2919. Springer, 2003, pp. 502–518.
- [10] S. A. Seshia, "Adaptive eager boolean encoding for arithmetic reasoning in verification," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-05-134, May 2005.
- [11] O. Shtrichman, "Tuning SAT checkers for bounded model checking," in *CAV '00: Proc. of the 12th International Conference on Computer Aided Verification*, ser. LNCS, vol. 1855. Springer, 2000, pp. 480–494.
- [12] O. Shacham and E. Zarpas, "Tuning the VSIDS Decision Heuristic for Bounded Model Checking," in *Fourth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions (MTV 2003), May 29-30, 2003, Hyatt Town Lake Hotel, Austin, Texas, USA*. IEEE Computer Society, 2003, pp. 75–79.
- [13] F. Lu, L.-C. Wang, K.-T. T. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," in *DAC '03: Proc. of the 40th conference on Design automation*. ACM Press, 2003, pp. 436–441.
- [14] J. C. Spall, *Introduction to Stochastic Search and Optimization*, ser. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., 2003.
- [15] B. Adenso-Diaz and M. Laguna, "Fine-tuning of algorithms using fractional experimental design and local search," *Operations Research*, vol. 54, no. 1, pp. 99–114, Jan–Feb 2006.
- [16] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp, "A racing algorithm for configuring metaheuristics," in *GECCO '02: Proc. of the Genetic and Evolutionary Computation Conference*, 2002, pp. 11–18.
- [17] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.
- [18] D. Babić and M. Musuvathi, "Modular Arithmetic Decision Procedure," Microsoft Research Redmond, Tech. Rep. TR-2005-114, 2005.
- [19] D. Babić and A. J. Hu, "Exploiting Shared Structure in Software Verification Conditions," in submission.
- [20] D. Jackson, "Automating first-order relational logic," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 130–139, 2000.
- [21] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated local search," in *Handbook of Metaheuristics*. Kluwer, 2002, pp. 321–353.
- [22] H. H. Hoos and T. Stützle, *Stochastic Local Search - Foundations & Applications*. Morgan Kaufmann, 2005.
- [23] E. Zarpas, "Benchmarking SAT Solvers for Bounded Model Checking," in *SAT '05: Proc. of the 8th International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 3569. Springer, 2005, pp. 340–354.
- [24] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown, "Performance prediction and automated tuning of randomized and parametric algorithms," in *CP '06: Proc. of the Twelfth International Conference on Principles and Practice of Constraint Programming*, 2006, pp. 213–228.
- [25] R. Battiti and M. Brunato, *Approximation Algorithms and Metaheuristics*. CRC Press, 2007, ch. 21: Reactive Search: Machine Learning for Memory-based Heuristics, pp. 21–1 – 21–17.

# Verifying Correctness of Transactional Memories

Ariel Cohen (CS/CIMS/NYU, arielc@cs.nyu.edu) John W. O’Leary (Intel, john.w.oleary@intel.com) Amir Pnueli (CS/CIMS/NYU, amir@cs.nyu.edu) Mark R. Tuttle (Intel, tuttle@acm.org) Lenore D. Zuck (CS/UIC, lenore@cs.uic.edu)

**Abstract**—We show how to verify the correctness of transactional memory implementations with a model checker. We show how to specify transactional memory in terms of the admissible interchange of transaction operations, and give proof rules for showing that an implementation satisfies this specification. This notion of an admissible interchange is a key to our ability to use a model checker, and lets us capture the various notions of transaction conflict as characterized by Scott. We demonstrate our work using the TLC model checker to verify several well-known implementations described abstractly in the  $TLA^+$  specification language.

**Index Terms**—Verification, transactional memory, model checking, HTM, STM,  $TLA^+$ , TLC.

## I. INTRODUCTION

The most important development in processor architecture in the last decade has been the shift from single-threaded, single-core processors to multi-threaded, multi-core processors. Taking advantage of these new processors, however, requires rewriting our applications as multi-threaded programs, and multi-threaded programs are hard to write, especially when several threads need to access the same data. Conventional approaches employ locks to regulate access to shared data, but locks are subtle and hard to use correctly. Some well-known problems with locks are *priority inversion*, which can occur when a low priority thread holds a lock needed by a higher priority thread; and *deadlock*, which can occur when several threads attempt to acquire the same set of locks in a different order.

*Transactional memory* [1] is a programming abstraction intended to simplify the synchronization of conflicting memory accesses (by concurrent threads) without the headaches associated with locks. A *transaction* is a sequence of memory operations that appears to be performed atomically with respect to other memory operations. The idea is that if a concurrent program is written so that each access to a shared data structure is encapsulated within a transaction, then all reads and writes to the data structure will appear to occur in isolation in some sequential order, and the established theory of database serializability will help us reason about the correctness of such programs. Early hardware implementations of transactional memory were limited to relatively small transactions, but recent software implementations (sometimes depending on limited hardware support) have managed to remove this restriction.

Larus and Rajwar [2] survey nearly 40 implementations of transactional memory in their comprehensive book on the subject, which differ in many dimensions. An implementation may employ *eager version control* (or *direct update*) in which a transaction modifies an object in place and restores the object to its original value upon abort, or may employ *lazy version control* (or *deferred update*) in which a transaction modifies a private copy of the object and overwrites the object with this private copy upon commit. An implementation may support *weak atomicity* or *strong atomicity* depending on whether the implementation guarantees transactional semantics only to object references within transactions or to all object references (even those outside of transactions). Different implementations may use very different approaches to detecting conflicts among transactions such

as *lazy or eager invalidation*, and support many different progress conditions in the presence of contention such as *wait freedom*, *lock freedom*, or *obstruction freedom*.

Scott [3] wrote a widely-cited paper that was the first to characterize transactional memory in a way that captured and clarified the many semantic distinctions among the most popular implementations. His approach was to begin with classical notions of transactional histories and sequential specifications, and to introduce two important notions. The first was a *conflict function* which specifies when two transactions cannot both succeed (a safety condition). The second was an *arbitration function* which specifies which of two transactions must fail (a liveness condition). Scott’s work went a long way toward making sense of transactional memory semantics, but his work was purely semantic and did not immediately facilitate mechanical verification of implementations.

In this paper, we present an abstract model for specifying transactional memory semantics inspired by Scott’s original work, and a proof rule for verifying that an implementation satisfies a transactional memory specification. The premisses of our proof rule can be checked with a model checker, and we demonstrate the method by modeling three well-known transactional memory implementations in  $TLA^+$  and proving their correctness with the model checker TLC. The essential contribution of this paper that enables mechanical checking is the notion of an *admissible interchange* used to model the approaches to conflict detection and resolution characterized by Scott in his paper. The work we report here is preliminary, but we hope it will form the basis for analysis of well-known issues like *privatization* and *granular lost update* in addition to implementation correctness, and for analysis of the interaction between hardware and software support for transactional memory.

The rest of this paper is organized as follows. We begin with preliminary definitions related to transactions and transaction sequences in Section II, and we define an admissible interchange in Section III. This definition is the key to our ability to model check transactional memory implementations, and we show how Scott’s transaction conflict classes can be characterized in terms of admissible interchanges. We give our specification of a transactional memory and what it means for an implementation to be correct in Section IV, and we give proof rules for verifying implementation correctness in Section V. We sketch the correctness proofs for several implementations of transactional memory in Section VI, and show how to use a model checker to verify their correctness in Section VII. Finally, in Section VIII, we end with some conclusions and open problems.

## II. TRANSACTIONAL SEQUENCES

Assume  $n$  clients that direct transactional requests to a *memory system*, denoted by *memory*. The requests that can be issued by client  $i$  are:

- $\blacktriangleleft_i$  – An open transaction request.
- $R_i(x)$  – A read request from address  $x \in \mathbb{N}$ .
- $W_i(y, v)$  – A request to write the value  $v \in \mathbb{N}$  to address  $y \in \mathbb{N}$ .
- $\blacktriangleright_i$  – A close transaction request.

The memory provides a response for each request. Responses are subject to the following constraints:

transaction) the memory returns an error flag. For requests that are accepted, and do not require a special response (e.g.,  $\blacktriangleleft_i$  when there is no pending  $i$  transaction), the memory responds with some acknowledgment. For accepted requests that require a response the memory provides a return value. For  $R_i(x)$ , it is a natural number indicating the value of the memory at location  $x$ . For  $\blacktriangleright_i$ , the memory responds with “commit” or “abort,” according to its decision on whether the transaction should be *committed* or *aborted*.

Let  $E_i: \{\blacktriangleleft_i, R_i(x, u), W_i(x, v), \blacktriangleright_i, \blacktriangleright_i^*\}$  be the set of *observable events* associated with client  $i$ , where  $\blacktriangleright_i^*$  represents the closing of a transaction that has been aborted (while  $\blacktriangleright_i$  represents the closing of a transaction that has been committed). We consider as observable only requests that are accepted, and we include the memory’s response for  $R_i(x)$  and  $\blacktriangleright_i$  requests (rather than the requests themselves). In this paper we also mandate that the order in which the memory issues its commit responses (and therefore the order of observable  $\blacktriangleright_i$  events) uniquely determines the order of committed transactions. Let  $E$  be the set of all observable events over all clients, i.e.,  $E = \bigcup_{i=1}^n E_i$ .

Note that we have defined  $R_i(x)$  to be the request corresponding to the response  $R_i(x, u)$ , and that we are abusing notation by writing  $\blacktriangleleft_i$ ,  $W_i(y, v)$ ,  $\blacktriangleright_i$  to denote both a request and a corresponding response when the meaning is clear from context. We will also denote responses  $R_i(x, u)$  and  $W_i(x, v)$  by  $R_i$  and  $W_i$  when the exact values of the parameters are unimportant or are clear from context.

Let  $\sigma: e_0, e_1, \dots, e_k$  be a finite sequence of observable  $E$ -events. The sequence  $\sigma$  is called a *well-formed transactional sequence* (TS for short) if the following conditions hold:

- 1) For every client  $i$ , let  $\sigma|_i$  be the sequence obtained by projecting  $\sigma$  onto  $E_i$ . Then  $\sigma|_i$  satisfies the regular expression  $T_i^*$ , where  $T_i$  is the regular expression  $\blacktriangleleft_i (R_i + W_i)^* (\blacktriangleright_i + \blacktriangleright_i^*)$ . For each occurrence of  $T_i$  in  $\sigma|_i$ , we refer to the first and last elements as *matching*. The notion of matching is lifted to  $\sigma$  itself, where  $\blacktriangleleft_i$  and  $\blacktriangleright_i$  (or  $\blacktriangleright_i^*$ ) are matching if they are matching in  $\sigma|_i$ ;
- 2) The sequence  $\sigma$  is *locally read-write consistent*: i.e., for any subsequence  $W_i(x, v)\eta R_i(x, u)$  in  $\sigma$ , where  $\eta$  contains no event of the form  $\blacktriangleright_i$ ,  $\blacktriangleright_i^*$ , or  $W_i(x, w)$ , we have  $u = v$ .

We denote by  $\mathcal{T}$  the set of all well-formed transactional sequences, and by  $\text{pref}(\mathcal{T})$  the set of prefixes of such sequences.

Notice that the requirement of local read-write consistency can be enforced by each client locally. To build on this observation, we assume that, within a single transaction, there is no  $R_i(x)$  following a  $W_i(x)$ , and there are no two reads or two writes to the same address. As a result, we can assume that the sequence of events constituting a single  $i$ -transaction has the form

$$\blacktriangleleft_i R_i(x_1, u_1) \cdots R_i(x_r, u_r) W_i(y_1, v_1) \cdots W_i(y_w, v_w) \{\blacktriangleright_i, \blacktriangleright_i^*\}$$

where the addresses in each of the sequences  $x_1, \dots, x_r$  and  $y_1, \dots, y_w$  are pairwise distinct. With this assumption, the requirement of local read-write consistency is always (vacuously) satisfied.

The TS  $\sigma$  is called *atomic* if:

- 1) It satisfies the regular expression  $(T_1 + \cdots + T_n)^*$ . That is, there is no overlap between any two transactions.
- 2) The sequence  $\sigma$  is *globally read-write consistent*: for any subsequence  $W_i(x, v)\eta R_j(x, u)$  in  $\sigma$ , where  $\eta$  contains  $\blacktriangleright_i$  but contains no event  $W_k(x, \cdot)$  followed by an event  $\blacktriangleright_k$ , it is the case that  $u = v$ .

### III. INTERCHANGING EVENTS

When is a TS  $\sigma$  a correct behavior of a transactional memory implementation? It is natural to say that  $\sigma$  is correct if it can be transformed into an atomic TS by first removing from it all

that belong to aborted transactions, then freely interchanging adjacent events that belong to committed transactions. This correctness criterion is known as *serializability*. Since we require that the order of  $\blacktriangleright_i$  events determines the order of committed transactions, we choose to disallow the interchange of  $\blacktriangleright$  events. This narrower criterion is known as *strict serializability*, and we will further refine it throughout the rest of this section.

Strict serializability, by itself, is far from a satisfactory correctness criterion for TM implementations. Let us say that transactions  $T_i$  and  $T_j$  overlap when  $\blacktriangleleft_i$  precedes  $\blacktriangleright_j$  and  $\blacktriangleleft_j$  precedes  $\blacktriangleright_i$ , and suppose we wish to specify a class of implementations that forbid two overlapping transactions to both commit. Strict serializability is much too generous a specification, as many strictly serializable transactional sequences contain overlapping transactions. Scott [3] introduced *conflicts* to describe the TS’s characteristic of different classes of implementations (in Scott’s terminology, our hypothetical class of implementations avoids *overlap conflicts*). We will describe conflicts by restricting which events can be exchanged during serialization. To specify the class of implementations that forbid overlapping transactions, for example, we will add the restriction that adjacent  $\blacktriangleleft$  and  $\blacktriangleright$  events cannot be interchanged during serialization: thus no TS with overlapping events will be strictly serializable.

Before introducing our notion of admissible interchanges, we briefly describe Scott’s six classes of conflicts. For a TS  $\sigma$ , let  $\prec_\sigma$  denote the precedence relation of events in  $\sigma$ , meaning that  $e_i \prec_\sigma e_j$  if  $e_i$  occurs before  $e_j$  in  $\sigma$ . We omit the  $\sigma$  subscript when its identity is clear from the context.

- 1) A TS  $\sigma$  has an *overlap conflict* if for some transactions  $T_i$  and  $T_j$ , we have  $\blacktriangleleft_i \prec \blacktriangleright_j$  and  $\blacktriangleleft_j \prec \blacktriangleright_i$ .
- 2) A TS  $\sigma$  has a *writer overlap conflict* if two transactions overlap and one performs a write before the other terminates, i.e., for some  $T_i$  and  $T_j$ , we have  $\blacktriangleleft_i \prec W_j \prec \blacktriangleright_i$  or  $W_j \prec \blacktriangleleft_i \prec \blacktriangleright_j$ .
- 3) A TS has a *lazy invalidation conflict* if commitment of one transaction may invalidate a read of the other, i.e., if for some transaction  $T_i$  and  $T_j$  and some memory address  $x$ , we have  $R_i(x), W_j(x) \prec \blacktriangleright_j \prec \blacktriangleright_i$ .
- 4) A TS has an *eager W-R conflict* if it has a lazy invalidation conflict, or if for some transactions  $T_i$  and  $T_j$  and some memory address  $x$ , we have  $W_i(x) \prec R_j(x) \prec \blacktriangleright_i$ .
- 5) A TS has a *mixed invalidation conflict* if it has a lazy invalidation conflict, or if for some transaction  $T_i$  and  $T_j$ , and some memory address  $x$ , we have  $R_i(x) \prec W_i(x), W_j(x) \prec \blacktriangleright_i, \blacktriangleright_j$ .
- 6) A TS has an *eager invalidation conflict* if it has an eager W-R conflict, or if for some transaction  $T_i$  and  $T_j$  and some memory address  $x$ , we have  $R_i(x) \prec W_j(x) \prec \blacktriangleright_i$ .

Let  $c$  be some conflict (e.g., “write overlap”). We denote by  $\mathcal{F}_c$  the *resolving predicate* describing the interchanges that may resolve a  $c$ -conflict. For a pair of events  $\langle e_i, e_j \rangle$  that belong to transactions  $T_i$  and  $T_j$  (where  $i \neq j$ ), we denote by  $\langle e_i, e_j \rangle \models \mathcal{F}_c$  the fact that  $\mathcal{F}_c$  implies that the interchange  $\langle e_i, e_j \rangle$  may resolve a  $c$ -conflict. In Fig. 1 we define  $\models \mathcal{F}_c$  for each of Scott’s conflicts  $c$  and every pair  $\langle e_i, e_j \rangle$ . In the full version of this paper we describe the language used to define the  $\mathcal{F}$ ’s.

Given a conflict  $c$  and the resolving predicate  $\mathcal{F}_c$  that corresponds to it, a TS  $\sigma$  is said to be *serializable* with respect to  $\mathcal{F}_c$  if it can be transformed into an atomic TS by a sequence of admissible interchanges (that do not satisfy  $\mathcal{F}_c$ ). Note that this definition is not equivalent to Scott’s definition of  $c$  which, in some cases, may imply interchanges that are not admissible (that is, that satisfy  $\mathcal{F}_c$ ).

The sequence  $\tilde{\sigma}$  is called the *purified version* of TS  $\sigma$  if  $\tilde{\sigma}$  is obtained by removing from  $\sigma$  all aborted transactions, i.e., removing the opening and closing events for such a transaction and all the read- and write events by the same client that occurred between the opening and

Conflict ( $c$ )	$\langle e_i, e_j \rangle \models \mathcal{F}_c$ if:
Overlap ( $o$ )	$e_i = \blacktriangleleft_i \wedge e_j = \blacktriangleright_j$
Writer Overlap ( $wo$ )	$\exists x, u. (e_i = \blacktriangleleft_i \wedge e_j = W_j(x, u) \vee e_i = W_i(x, u) \wedge e_j \in \{\blacktriangleleft_j, \blacktriangleright_j\})$
Lazy Invalidation ( $li$ )	$\exists x, u, v. (W_j(x, u) \in T_j \wedge e_i = R_i(x, v) \wedge e_j = \blacktriangleright_j)$
Eager W-R ( $ewr$ )	$\mathcal{F}_i \vee (\exists x, u, v. e_i = W_i(x, u) \wedge e_j = R_j(x, v))$
Mixed Invalidation ( $mi$ )	$\mathcal{F}_i \vee \exists x, u, v. (e_i = R_i(x) \wedge e_j = W_j(x) \wedge e_i \prec W_i(x, u) \prec \blacktriangleright_j \wedge e_j \prec \blacktriangleleft_i \vee e_i = W_j(x, u) \wedge e_j = \blacktriangleright_j \wedge R_i(x, u) \prec W_i(x))$
Eager Invalidation ( $ei$ )	$\mathcal{F}_{ewr} \vee \exists x, u, v. (e_i = R_i(x, u) \wedge e_j = W_j(x, v) \wedge e_j \prec \blacktriangleleft_i \vee e_i = W_i(x, u) \wedge e_j = \blacktriangleright_j \wedge R_j(x, v) \prec e_i)$

Fig. 1. Conflicts and Their Corresponding Predicates

closing events. When we specify the correctness of a transactional memory implementation, only the purified versions of the implementation's transaction sequences will have to be serializable.

#### IV. TM: SPECIFICATION AND IMPLEMENTATION

Let  $\mathcal{F}$  be a resolving predicate which we fix for the remainder of this section. We now describe  $Spec_{\mathcal{F}}$  – a specification of transactional memory that generates all TSs serializable with respect to  $\mathcal{F}$  and a definition of a correct implementation of  $Spec_{\mathcal{F}}$ .

The specification  $Spec_{\mathcal{F}}$  can be formally presented as an FDS (fair transition system, see Appendix). It uses the following data structures:

- $spec\_mem: \mathbb{N} \mapsto \mathbb{N}$  — A persistent memory, represented as an array of naturals. For simplicity, we represent it as an infinite array. Initially, for every  $i \geq 0$ ,  $spec\_mem[i] = 0$ ;
- $q$ : **queue of**  $E \cup \bigcup_{i=1}^n \{mark_i\}$  — A queue of pending events, initially empty;
- $spec\_out$ : scalar in  $E_{\perp} = E \cup \{\perp\}$  — an output variable recording responses to clients, initially  $\perp$ ;
- $doomed$ : **array**  $[1..n]$  **of booleans** — An array recording which transactions are doomed to be aborted. Initially  $doomed[i] = \text{F}$  for every  $i$ .

Let

$tr: \blacktriangleleft_i R_i(x_1, u_1), \dots, R_i(x_r, u_r), W_i(y_1, v_1), \dots, W_i(y_w, v_w) \blacktriangleright_i$

be a transaction. We say that  $tr$  is *consistent* with  $spec\_mem$  if, for each  $j \in [1..r]$ ,  $spec\_mem[x_j] = u_j$ . The *update* of  $spec\_mem$  by  $tr$  is defined to be the memory  $spec\_mem'$  such that, for each  $j \in [1..w]$ ,  $spec\_mem'[y_j] = v_j$  and, for all  $k \notin \{y_1, \dots, y_w\}$ ,  $spec\_mem'[k] = spec\_mem[k]$ .

Intuitively, the stream of  $spec\_out$ 's is the sequence of observable events. Pending transactions are partitioned to two categories. *Active* transactions, whose events are maintained in  $q$  in the order they are in  $spec\_out$ , and *doomed* transactions, that must be aborted, indicated by  $doomed[i] = \text{T}$ . When a transaction is doomed, all its events are removed from  $q$ , and subsequent events are echoes by  $spec\_out$  but nowhere stored. When a pending transaction is committed, aborted, or doomed, all its events (which may be none if the transaction is doomed) are removed from  $q$ , and subsequent events are stored nowhere and it is marked as “undoomed.” A transaction  $T_i$  is *doomed* if  $doomed[i] = \text{T}$ ;  $T_i$  is *active* if  $q$  has some  $E_i$ -event;  $T_i$  is *inactive* if its neither active nor doomed.

For every active transaction  $T_i$ , we allow the queue  $q$  to include a special symbol,  $mark_i$ . The symbol  $mark_i$  is added to the queue when a  $T_i$  issues a close request, and some tests are done to determine whether it can safely close. If the test is successful,  $spec\_out$  is set to  $\blacktriangleright_i$ , otherwise, it is set to  $\blacktriangleleft_i$ , and then  $mark_i$  as well as all the  $E_i$ -events are removed from the queue. We say that  $q$  is marked (unmarked) if it has some (no)  $mark_i$  symbol.

Transaction  $a_1$ – $a_5$  are applicable only when  $q$  is unmarked. Note that  $a_4$  and  $a_5$  do not set  $spec\_out$  to a value. For such cases we assume that  $spec\_out$  is set to  $\perp$ .

- $a_1$ . For some  $i \in [1..n]$ , if  $T_i$  is inactive, write  $\blacktriangleleft_i$  to  $spec\_out$  and append it to the end of the queue  $q$ .
- $a_2$ . For some  $i \in [1..n]$ , and  $x, u \in \mathbb{N}$ , if  $T_i$  is active or doomed, write  $W_i(x, u)$  to  $spec\_out$ . If  $T_i$  is active, then  $W_i(x, u)$  is appended to the end of the queue  $q$ .
- $a_3$ . For some  $i \in [1..n]$ , and  $x, u \in \mathbb{N}$ , if  $T_i$  is active or doomed, write  $R_i(x, u)$  to  $spec\_out$ . If  $T_i$  is active, then  $R_i(x, u)$  is appended to  $q$ . Moreover, in this case we also require that the events of  $T_i$  are locally consistent.
- $a_4$ . For some  $i \in [1..n]$  such that  $T_i$  is active, remove all of events in  $E_i$  from the queue  $q$  and set  $doomed[i]$  to  $\text{T}$ .
- $a_5$  For some  $i \in [1..n]$  such that  $T_i$  is active, add  $mark_i$  to the end of  $q$ .

Transition  $a_6$ – $a_8$  deal with commits and aborts. It is  $a_7$  that determines whether a transaction marked for commit can indeed commit.

- $a_6$ . For some  $i \in [1..n]$  such that  $T_i$  is active or doomed, write  $\blacktriangleright_i$  to  $spec\_out$ , and remove all of  $E_i$ - and  $mark_i$ -events from the queue  $q$ , and set  $doomed[i]$  to  $\text{F}$ .
- $a_7$ . For some  $i \in [1..n]$  such that  $T_i$  is active, if  $T_i$  is consistent with  $spec\_out$ , all of its events appear consecutively in the front of  $q$ , and  $mark_i$  is in  $q$ , then write  $\blacktriangleright_i$  to  $spec\_out$ , update  $spec\_mem$  according to  $T_i$ , and remove all  $E_i$ - and  $mark_i$ -events from the queue.
- $a_8$ . Interchange the order of two contiguous events  $e_i, e_j$  in  $q$  belonging to different transactions  $T_i$  and  $T_j$ , respectively, if  $mark_j$  is in  $q$ , and  $\langle e_i, e_j \rangle \not\models \mathcal{F}$ . We treat  $mark_j$  as if it is a  $\blacktriangleright_j$  and assume a hypothetical  $\blacktriangleright_i$  appended at the end of  $q$ .
- $a_9$ . An idling transition which does not modify  $spec\_mem$ ,  $q$  or  $doomed$ .

Note that the updates of the queue in  $a_4$ ,  $a_6$ , and  $a_7$  are not standard queue operations.

The specification has  $n$  associated justice requirements, namely, for every  $i = 1, \dots, n$ :

there are infinitely many states in which  $q|_i$  is empty.

A sequence  $\sigma$  over  $E^*$  is *compatible* with  $Spec_{\mathcal{F}}$  if  $\sigma$  can be obtained by the sequence of  $spec\_out$  which  $Spec_{\mathcal{F}}$  outputs, once all the  $\perp$ 's are removed. We then have:

*Claim 1:* For every sequence  $\sigma$  over  $E$ ,  $\sigma$  is compatible with  $Spec_{\mathcal{F}}$  iff  $\sigma$  is serializable with respect to  $\mathcal{F}$ .

An *implementation TM*: ( $read, close$ ) of a transactional memory consists of a pair of functions

$$\begin{aligned} read & : \text{pref}(T) \times [1..n] \times \mathbb{N} \rightarrow \mathbb{N} \quad \text{and} \\ close & : \text{pref}(T) \times [1..n] \rightarrow \{\text{commit}, \text{abort}\} \end{aligned}$$

For a prefix  $\sigma$  of a TS,  $read(\sigma, i, x)$  is the response (value) of the memory to an accepted  $R_i(x)$  request immediately following  $\sigma$ , and  $close(\sigma, i)$  is the response (*commit* or *abort*) of the memory to a  $\blacktriangleright_i$  request immediately following  $\sigma$ .

39  $\in \mathcal{T}$  is said to be *compatible* with the memory  $TM$  if:

- 1) For every prefix  $\eta Ri(x, u)$  of  $\sigma$ ,  $read(\eta, i, x) = u$ .
- 2) For every prefix  $\eta \blacktriangleright_i$  of  $\sigma$ ,  $close(\eta, i) = commit$ .
- 3) For every prefix  $\eta \blacktriangleright_i$  of  $\sigma$ ,  $close(\eta, i) = abort$ .

An implementation  $TM: (read, close)$  is a *correct implementation of a transactional memory with respect to  $\mathcal{F}$*  if every TS compatible with  $TM$  is also compatible with  $Spec_{\mathcal{F}}$ .

## V. VERIFYING IMPLEMENTATION CORRECTNESS

In this section we present proof rules for verifying that an implementation satisfies the specification  $Spec$ . The approach is an adapted version of the rule presented in [4].

To apply the underlying theory, we assume that both the implementation and the specifications are represented as a *fair discrete system* (FDS) of the form  $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$ . We refer the reader to the appendix for additional details about this presentation of reactive systems.

In the current application, we prefer to adopt an *event-based view* of reactive systems, by which the observed behavior of a system is a (potentially infinite) set of events. Technically, this implies that the set of observable variables consists of a single variable  $\mathcal{O}$ , to which we refer as the *output variable*. It is also required that the domain of  $\mathcal{O}$  always includes the value  $\perp$ , implying no observable event. In our case, the domain of the output variable is  $E_{\perp} = E \cup \{\perp\}$ .

Let  $\eta : e_0, e_1, \dots$  be an infinite sequence of  $E_{\perp}$ -values. The  $E_{\perp}$ -sequence  $\tilde{\eta}$  is called a *stuttering variant* of the sequence  $\eta$  if it can be obtained by removing or inserting finite strings of the form  $\perp, \dots, \perp$  at (potentially infinitely many) different positions within  $\eta$ .

Let  $\sigma : s_0, s_1, \dots$  be a computation of FDS  $\mathcal{D}$ . The *observation* corresponding to  $\sigma$  is the  $E_{\perp}$  sequence  $s_0[\mathcal{O}], s_1[\mathcal{O}], \dots$  obtained by listing the values of the output variable  $\mathcal{O}$  in each of the states. We denote by  $Obs(\mathcal{D})$  the set of all observations of system  $\mathcal{D}$ .

Let  $\mathcal{D}_C$  and  $\mathcal{D}_A$  be two systems, to which we refer as the *concrete* and *abstract* systems, respectively. We say that system  $\mathcal{D}_A$  *abstracts* system  $\mathcal{D}_C$  (equivalently  $\mathcal{D}_C$  *refines*  $\mathcal{D}_A$ ), denoted  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$  if, for every observation  $\eta \in Obs(\mathcal{D}_C)$ , there exists  $\tilde{\eta} \in Obs(\mathcal{D}_A)$ , such that  $\tilde{\eta}$  is a stuttering variant of  $\eta$ . In other words, modulo stuttering,  $Obs(\mathcal{D}_C)$  is a subset of  $Obs(\mathcal{D}_A)$ .

### A. A Verification Rule Based on Abstraction Mapping

Based on the *abstraction mapping* of [5], we present in Fig. 2 a proof rule that reduces the abstraction problem into a verification problem. There, we assume two comparable FDS's, a *concrete*  $\mathcal{D}_C : \langle V_C, \mathcal{O}_C, \Theta_C, \rho_C, \mathcal{J}_C, \mathcal{C}_C \rangle$  and an *abstract*  $\mathcal{D}_A : \langle V_A, \mathcal{O}_A, \Theta_A, \rho_A, \mathcal{J}_A, \mathcal{C}_A \rangle$ , and we wish to establish that  $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ . Without loss of generality, we assume that  $V_C \cap V_A = \emptyset$ , and that there exists a 1-1 correspondence between the concrete observables  $\mathcal{O}_C$  and the abstract observables  $\mathcal{O}_A$ .

The method assumes the identification of an *abstraction mapping*  $\alpha : (V_A = \mathcal{E}^{\alpha}(V_C))$  which assigns to each abstract variable  $X \in V_A$  an expression  $\mathcal{E}_X^{\alpha}$  over the concrete variables  $V_C$ . For an abstract assertion  $\varphi$ , we denote by  $\varphi[\alpha]$  the assertion obtained by replacing each abstract variable  $X \in V_A$  by its concrete expression  $\mathcal{E}_X^{\alpha}$ . We say that the abstract state  $S$  is an  $\alpha$ -image of the concrete state  $s$ , written  $S = \alpha(s)$ , if the values of  $\mathcal{E}^{\alpha}$  in  $s$  equal the values of the variables  $V_A$  in  $S$ .

Premise A1 of the rule states that if  $s$  is a concrete initial state, then  $S = \alpha(s)$  is an initial abstract state.

Premise A2 states that if concrete state  $s_2$  is a  $\rho_C$ -successor of concrete state  $s_1$ , then the abstract state  $S_2 = \alpha(s_2)$  is a  $\rho_A$ -successor of  $S_1 = \alpha(s_1)$ . The box  $(\ )$  is the (linear time) temporal operator for “from here onwards.” Together, A1 and A2 guarantee that, for every run  $s_0, s_1, \dots$  of  $\mathcal{D}_C$  there exists a run  $S_0$

of  $\mathcal{D}_A$ , such that  $S_j = \alpha(s_j)$  for every  $j \geq 0$ . Premise A3 states that the observables of the concrete state  $s$  and its  $\alpha$ -image  $S = \alpha(s)$  are equal. Premises A4 and A5 ensure that the abstract fairness requirements (justice and compassion, respectively) hold in any abstract state sequence which is a (point-wise)  $\alpha$ -image of a concrete computation. Here,  $(\ )$  is the (linear time) temporal operator for “eventually,” thus,  $(\ )^{\infty}$  means “infinitely often.” It follows that every  $\alpha$ -image of a concrete computation  $\sigma$  obtained by applications of premises A1 and A2 is an abstract computation whose observables match the observables of  $\sigma$ . This leads to the following claim:

*Claim 2:* If the premises of rule ABS-MAP are valid for some choice of  $\alpha$ , then  $\mathcal{D}_A$  is an abstraction of  $\mathcal{D}_C$ .

### B. A Rule Based on an Abstraction Relation

It is not always possible to relate abstract to concrete states by a functional correspondence which maps each concrete state to a unique abstract state. In many cases, we cannot find an abstraction mapping, but can identify an *abstraction relation*  $R(V_C, V_A)$  (which induces a relation  $R(s, S)$ ).

In Fig. 3, we present proof rule ABS-REL which only assume an abstraction relation between the concrete and abstract states.

Premise R2 of the rule allows a single concrete transition to be emulated by a sequence of abstract transitions. This is done via the transitive closure  $\rho_A^+$  which is defined as follows:

Let  $S_0, S_1, \dots, S_k, k > 0$ , be a sequence of abstract states, such that  $\langle S_i, S_{i+1} \rangle \models \rho_A$  for every  $i \in [0..k-1]$ , and for some  $\ell \in [1..k]$ , for every  $i \in [1..k]$ , if  $i \neq \ell$  then  $S_i[\mathcal{O}] = \perp$ . Then  $\langle S_0, \tilde{S}_k \rangle \models \rho_A^+$ , where  $\tilde{S}_k = S_k[\mathcal{O} := S_{\ell}[\mathcal{O}]]$  is obtained from  $S_k$  by assigning the variable  $\mathcal{O}$  (the single output variable) the value that it has in state  $S_{\ell}$ . This definition allows to perform first some “setting up” transitions that have no externally observable events, followed by a transition that produces a non-trivial observable value, followed by a finite number of “clean-up” transitions. The observable effect of the composite transition is taken to be the observable output of the only observable transition in the sequence.

Premise R1 of the rule states that for every initial concrete state  $s$ , it is possible to find an initial abstract state  $S \models \Theta_A$ , such that  $\langle s, S \rangle \models R$ .

Premise R2 states that for every pair of concrete states,  $s_1$  and  $s_2$ , such that  $s_2$  is a  $\rho_C$ -successor of  $s_1$ , and an abstract state  $S_1$  which is a  $R$ -related to  $s_1$ , it is possible to find an abstract state  $S_2$  such that  $S_2$  is  $R$ -related to  $s_2$  and is also a  $\rho_A^+$ -successor of  $S_1$ . Together, R1 and R2 guarantee that, for every run  $s_0, s_1, \dots$  of  $\mathcal{D}_C$  there exists a run  $S_0, \dots, S_{i_1}, \dots, S_{i_2}, \dots$ , of  $\mathcal{D}_A$ , such that for every  $j \geq 0$ ,  $S_{i_j}$  is  $R$ -related to  $s_j$  and all abstract states  $S_k$ , for  $i_j < k < i_{j+1}$ , have no observable variables. Premise R3 states that if abstract state  $S$  is  $R$ -related to the concrete state  $s$ , then the two states agree on the values of their observables. Premises R4 and R5 ensure that the abstract fairness requirements (justice and compassion, respectively) hold in any abstract state sequence which is a (point-wise)  $R$ -related to a concrete computation. It follows that every sequence of abstract states which is  $R$ -related to a concrete computation  $\sigma$  and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of  $\sigma$ . This leads to the following claim:

*Claim 3:* If the premises of rule ABS-REL are valid for some choice of  $R$ , then  $\mathcal{D}_A$  is an abstraction of  $\mathcal{D}_C$ .

## VI. TRANSACTIONAL MEMORY IMPLEMENTATIONS

We now demonstrate how our proof rules can be used to verify the popular transactional memory implementations. Larusson and Alami [2] classify transactional memory implementations in

<b>A1.</b>	$\Theta_C \rightarrow \Theta_A[\alpha]$	
<b>A2.</b>	$\mathcal{D}_C \models (\rho_C \rightarrow \rho_A[\alpha][\alpha'])$	
<b>A3.</b>	$\mathcal{D}_C \models (\mathcal{O}_C = \mathcal{O}_A[\alpha])$	
<b>A4.</b>	$\mathcal{D}_C \models J[\alpha],$	for every $J \in \mathcal{J}_A$
<b>A5.</b>	$\mathcal{D}_C \models p[\alpha] \rightarrow q[\alpha],$	for every $(p, q) \in \mathcal{C}_A$
$\mathcal{D}_C \sqsubseteq \mathcal{D}_A$		

Fig. 2. Rule ABS-MAP.

<b>R1.</b>	$\Theta_C \rightarrow \exists V_A : R \wedge \Theta_A$	
<b>R2.</b>	$\mathcal{D}_C \models (R \wedge \rho_C \rightarrow \exists V'_A : R' \wedge \rho_A^+)$	
<b>R3.</b>	$\mathcal{D}_C \models (R \rightarrow \mathcal{O}_C = \mathcal{O}_A)$	
<b>R4.</b>	$\mathcal{D}_C \models (\forall V_A : R \rightarrow J),$	for every $J \in \mathcal{J}_A$
<b>R5.</b>	$\mathcal{D}_C \models (\exists V_A : R \wedge p) \rightarrow (\forall V_A : R \rightarrow q),$	for every $(p, q) \in \mathcal{C}_A$
$\mathcal{D}_C \sqsubseteq \mathcal{D}_A$		

Fig. 3. Rule ABS-REL.

terms of several properties. We focus on two of these properties, *conflict detection* and *version control*, both of which can be either “eager” or “lazy,” depending when conflicts are detected and when the memory is updated. Since one cannot have eager version management with lazy conflict detection, there are three possibilities left. We give a detail description of the proof of the lazy conflict detection and lazy version control, and sketch the remaining two.

#### A. Lazy Conflict Detection, Lazy Version Control

Denote this class by **II**. A representative of this class is TCC [6], and we give a simple implementation from this class that we refer to as  $TM_1$ .

The implementation uses the following data structures:

- $imp\_mem: \mathbb{N} \rightarrow \mathbb{N} \times A$  — A persistent memory. Initially,  $imp\_mem[j] = 0$  for all  $j \in \mathbb{N}$ ;
- $trans$ : **array[1..n] of list of  $E$**  — An array of lists. For each  $i \in [1..n]$ ,  $trans[i]$  is a sequence over  $E_i$  that lists the events of the currently pending transaction of client  $i$ , if such exists. Initially, every  $trans[i]$  is empty;
- $imp\_out$ : scalar in  $E_\perp = E \cup \{\perp\}$  — an output variable recording responses to clients, initially  $\perp$ .

The implementation reacts to possible requests by the clients. It accepts a request of  $\blacktriangleleft_i$  (“open transaction”), and rejects any other request if  $trans[i]$  is empty. An accepted  $R_i(x)$  request is responded by  $u$ , where  $u$  is such that  $W_i(x, u)$  is the last  $W_i(x)$  event in  $trans[i]$ , or, if no such event exists, by  $imp\_mem[x]$ ; Upon an accepted  $\blacktriangleright_i$  request,  $TM_1$  checks whether the transaction  $trans[i]$  is consistent with  $imp\_mem$ . If it is,  $TM_1$  returns to Client  $i$  a “commit”, updates  $imp\_mem$  according to  $trans[i]$ , and resets  $trans[i]$  to be empty. If  $trans[i]$  is not consistent with  $imp\_mem$ ,  $TM_1$  returns an “abort,” and resets  $trans[i]$  to empty.

Finally, the events corresponding to accepted requests are written to  $imp\_out$ , which is set to  $\perp$  with steps that don’t produce a response. Each of these events (with the exception of  $\blacktriangleright$  and  $\blacktriangleright_i$ ), is appended to the appropriate  $trans[i]$ .

The specification, described in Section IV, specifies not only the behavior of the Transactional Memory but also the combined behavior of the memory when coupled with a typical clients module. A generic clients module,  $Clients(n)$ , may, at any step, issue the next request for client  $i$ ,  $i \in [1..n]$ , provided the sequence of  $E_i$ -events issued so far (including the current one) forms a prefix of a well-formed TS. The justice requirement of  $Clients(n)$  is that eventually, every transaction must be closed by issuing a  $\blacktriangleright_i$ -request.

Combining modules  $TM_1$  and  $Clients(n)$  we obtain the complete implementation, defined by:

$$Imp_1 : TM_1 \parallel\parallel Clients(n)$$

where  $\parallel\parallel$  denote the *synchronous* composition operator defined in the appendix. We interpret this composition in a way that combines several of the actions of each of the modules into a single transition.

The possible actions of  $Imp_1$  are the following:

- $t_1$ . Set  $imp\_out = trans[i] = \blacktriangleleft_i$  if  $trans[i] = \Lambda$ ;
- $t_2$ . Set  $imp\_out$  to  $R_i(x, u)$  and append it to  $trans[i]$  if  $trans[i]$  is non-empty, and the last  $W_i(x)$  event in it is  $W_i(x, u)$ , or if  $trans[i]$  contains no  $W_i(x)$  event and  $u = imp\_mem[x]$ ;
- $t_3$ . Set  $imp\_out$  to  $W_i(y, v)$  and append  $W_i(y, v)$  to the end of  $trans[i]$  if  $trans[i]$  is non-empty;
- $t_4$ . Set  $imp\_out$  to  $\blacktriangleright_i$ , update  $imp\_mem$  according to  $trans[i]$ , and reset  $trans[i]$  to empty if  $trans[i]$  is non-empty and consistent with  $imp\_mem$ ;
- $t_5$ . Set  $imp\_out$  to  $\blacktriangleright_i$  and set  $trans[i]$  to empty if  $trans[i]$  is non-empty and is inconsistent with  $imp\_mem$ ;
- $t_6$ . Set  $imp\_out$  to  $\perp$  and leave all other variables unchanged.

Since  $Clients(n)$ ’s justice requires every transaction to eventually issue a  $\blacktriangleright$  request, and since  $t_4$  and  $t_5$  guarantee that each  $\blacktriangleright$  request empties the corresponding  $trans[i]$ , it follows that module  $Imp_1$  has a justice requirement: for each  $i = 1, \dots, n$ ,  $trans[i]$  is empty infinitely many times.

We now sketch a proof, using Rule ABS-REL, that  $Imp_1 \sqsubseteq Spec$ .

The application of rule ABS-REL requires the identification of a relation  $R$  which holds between concrete and abstract states. We use the relation  $R$  defined by:

$$spec\_out = imp\_out \wedge spec\_mem = imp\_mem \wedge \bigwedge_{i=1}^n (q_i = trans[i])$$

The relation  $R$  stipulates equality between  $spec\_out$  and  $imp\_out$  – the output of the implementation, and between  $spec\_mem$  and  $imp\_mem$ , and that, for each  $i \in [1..n]$ , the projection of  $q$  on the set of events pertinent to Client  $i$  equals  $trans[i]$ .

To simplify the proof, we assume (see the end of Section II) that all transactions have the form

$$(x_1, u_1) \cdots R_i(x_r, u_r) W_i(y_1, v_1) \cdots W_i(y_w, v_w) \{\blacktriangleright_i, \blacktriangleright_i\}$$

It is not difficult to see that premise R1 of rule ABS-REL holds, since the two initial conditions are given by

$$\begin{aligned} \Theta_C : \quad & \text{imp\_out} = \perp \wedge \text{imp\_mem} = \lambda i.0 \wedge \bigwedge_{i=1}^n (\text{trans}[i] = \Lambda) \\ \Theta_A : \quad & \text{spec\_out} = \perp \wedge \text{spec\_mem} = \lambda i.0 \wedge q = \Lambda \end{aligned}$$

and the relation  $R$  guarantees equality between the relevant variables.

The  $R$ -conjunct  $\text{spec\_out} = \text{imp\_out}$  guarantees the validity of premise R3.

We will now examine the validity of premise R2. This can be done by considering each of the concrete transitions  $t_1, \dots, t_6$ .

- $t_1$ . Transition  $t_1$  appends the event  $\blacktriangleleft_i$  to an empty  $\text{trans}[i]$  and outputs it to  $\text{imp\_out}$ . This can be emulated by an instance of abstract transition  $a_1$  which output  $\blacktriangleleft_i$  to  $\text{spec\_out}$  and places this event at the end of  $q$ . It can be checked that this joint action preserves the relation  $R$ , in particular, the relevant conjunct  $\bigwedge_{j=1}^n (q|_j = \text{trans}[j])$ .
- $t_2$ . Transition  $t_2$  appends to  $\text{trans}[i]$  (and outputs) the event  $R_i(x, u)$  where, due to the simplifying assumption,  $u = \text{imp\_mem}[x]$ . This can be matched by another instance of abstract transition  $a_2$ .
- $t_3$ . Transition  $t_3$  appends to  $\text{trans}[i]$  (and outputs) the event  $W_i(y, v)$ , which is matched by an instance of abstract transition  $a_2$ .
- $t_4$ . Transition  $t_4$  closes and commits the current transaction contained in  $\text{trans}[i]$  while outputting the event  $\blacktriangleright_i$ . This is possible if the transaction pending in  $\text{trans}[i]$  is consistent with  $\text{imp\_mem}$ . The transition also updates  $\text{imp\_mem}$  according to  $\text{trans}[i]$ , and then clears  $\text{trans}[i]$ .  
The emulation of this transition begins by the instance of  $a_5$  which appends  $\text{mark}_i$  to  $q$ , followed by a sequence of applications of abstract transition  $a_8$  which attempts to move all the elements of  $\text{trans}[i]$  to the front of the queue  $q$ , where  $\mathcal{F}$  is the trivial predicate  $F$  (thus, allowing any interchange). If successful, we apply abstract transition  $a_7$  which confirms that  $\text{trans}[i]$  is consistent with  $\text{spec\_mem}$  (must be true due to the  $R$ -conjunct  $\text{spec\_mem} = \text{imp\_mem}$ ), updates  $\text{spec\_mem}$  according to  $\text{trans}[i]$  (thus making it again equal to  $\text{imp\_mem}$ ), and removes all elements of  $\text{trans}[i]$  from  $q$ , thus reestablishing the  $R$ -conjunct  $\bigwedge_{j=1}^n (q|_j = \text{trans}[j])$ .
- $t_5$ . Transition  $t_5$  closes and aborts the transaction pending in  $\text{trans}[i]$  while outputting the event  $\blacktriangleright_i$ . This is possible only if the transaction pending in  $\text{trans}[i]$  is inconsistent with  $\text{imp\_mem}$ . The transition also clears  $\text{trans}[i]$ .  
The transition  $t_5$  is matched with the abstract transition  $a_6$  which outputs the event  $\blacktriangleright_i$  and removes from  $q$  all elements of the aborted transaction  $\text{trans}[i]$ . Note that  $\text{Spec}$  does not require an aborted transaction to be “uncommitable,” thus, we don’t have to (though we can) ensure that  $\text{Spec}$  cannot commit  $\text{trans}[i]$ .
- $t_6$ . The idling concrete transition  $t_6$  may be emulated by the idling abstract transition  $a_9$ .

It remains to verify premise R4. This premise requires showing that any concrete computation visits infinitely many times states satisfying  $\forall V_A : R \rightarrow J_A$ , where  $J_i : q|i = \Lambda$ , characterizes the set of abstract states in which the queue contains no  $E_i$  event. Since  $R$  requires that  $q|i = \text{trans}[i]$ , we obtain that Premise R4 is valid.

Premise R5 is vacuously valid since  $\text{Spec}$  has no compassion requirements.

Note that ABS-MAP does not suffice to construct step  $t_4$ , where the power of ABS-REL is demonstrated. We obtained a similar proof for a bounded instantiation using TLC, however, there  $\text{Spec}$  is defined as performing “meta-steps,” without which TLC, that uses an  $\text{AF}^*$ -like rule, cannot construct the relations ABS-REL does.

## B. Eager Conflict Detection, Lazy Version Control

Denote this class by **el**. A representative of **el** is LTM of [7]. Its definition of “conflict” is slightly stronger than “eager invalidation” by having writes to the same object as a conflict, thus, its forbidden interchange set consists of  $\mathcal{F}_{ei}$  and all pairs of the form  $(W_i(x), W_j(x))$ . In case of a conflict, the transaction that requests the second “offensive” memory access is aborted.

The main difference between **el** and the prior implementation  $TM_1$  is the conflict detection: upon receiving a  $R_i(x)$  such that  $W_j(x)$  is in some open transaction, or a  $W_i(x, v)$  such that  $W_j(x)$  or  $R_j(x)$  is in some open transaction, the transaction of client  $i$  is aborted. The system performs two steps – the first returns the result of the operation, and the second aborts the transaction. Thus, an abort is not only a possible response to a non-close transaction request, but every transaction that requests to be closed is committed. For our higher level description of this implementation, we add a new variable  $\text{toabort} \in [0..n]$ , that holds the id of the client whose transaction is to be aborted (0 indicates no such client exists).

The combination of an **el** memory and  $\text{Clients}(n)$  is the module **Iel** whose possible actions are:

- $t_1$ . If  $\text{toabort} = i > 0$ , then set  $\text{imp\_out}$  to  $\blacktriangleright_i$ , empty  $\text{trans}[i]$ , and set  $\text{toabort}$  to 0;

Else, do one of the following:

- $t_2$ . Set  $\text{imp\_out} = \text{trans}[i] = \blacktriangleleft_i$  if  $\text{trans}[i]$  is empty;
- $t_3$ . Set  $\text{imp\_out}$  to  $R(x, u)$ , and append it to  $\text{trans}[i]$ , if  $\text{trans}[i] \neq \Lambda$ ,  $u = \text{imp\_mem}[x]$  or  $W_i(x) \in \text{trans}[i]$  and the most recent such event is  $W_i(x, u)$ , and for every  $j \neq i$ ,  $W(x) \notin \text{trans}[j]$ ;
- $t_4$ . Set  $\text{imp\_out}$  to  $R(x, \text{imp\_mem}[u])$ , append it to  $\text{trans}[i]$ , and set  $\text{toabort}$  to  $i$  if  $\text{trans}[i] \neq \Lambda$ , and for some  $j \neq i$ , or  $W_j(x) \in \text{trans}[j]$ ;
- $t_5$ . Set  $\text{imp\_out}$  to  $W_i(x, v)$  and append it to  $\text{trans}[i]$ , if  $\text{trans}[i] \neq \Lambda$  and for every  $j \neq i$ ,  $W(x), R(x) \notin \text{trans}[j]$ ;
- $t_6$ . Set  $\text{imp\_out}$  to  $W_i(x, v)$ , append it to  $\text{trans}[i]$ , and set  $\text{toabort}$  to  $i$ , if  $\text{trans}[i] \neq \Lambda$  and for some  $j \neq i$ ,  $R_j(x)$  or  $W_j(x) \in \text{trans}[j]$ ;
- $t_7$ . Set  $\text{imp\_out}$  to  $\blacktriangleright_i$ , update  $\text{imp\_mem}$  according to  $\text{trans}[i]$  and reset  $\text{trans}[i]$  to  $\Lambda$ , if  $\text{trans}[i]$  is not empty;
- $t_8$ . Set  $\text{imp\_out}$  to  $\perp$  and leave all other variables unchanged;

Module **Iel** has a justice requirement for each  $i = 1, \dots, n$ , requiring that  $\text{trans}[i] = \Lambda$  infinitely many times.

To prove that **Iel** satisfies the specifications of Section IV, we use the same  $R$  used to verify  $TM_1$ , with respect to the admissible interchange associated with **el**.

STM of [8] is also an **el** implementation. There, clients must first obtain write locks on all memory locations they are likely to access in a transaction (the locks are requested in increasing order, to avoid deadlocks), which are released when the transaction completes. The locking mechanism can be accomplished by adding to each memory location an “owner” in the range  $[0..n]$  indicating which client currently has a write-lock on it, and refining **Iel** to accommodate the needs of STM.

## C. Eager Conflict, Eager Version Control

Denote this class by **ee**. A representative of **ee** is LogTM of [9]. Its definition of “conflict” and their resolution are exactly like those of **el**. Being eager-version, however, **ee** protocols update the memory upon a write. If later it is necessary to abort the transaction, then the memory is rolled back to its previous value. Since the protocol allows for more than one overlapping write, there is no need to know any information but the previous value of  $W$ ’s in pending

transactions. To thus add a set  $committed \subseteq n \times \mathbb{N} \times \mathbb{N}$  where  $n$  is a client id.  $committed$  stores, for every memory address  $x$  that was written by a currently pending transaction, the previous value written to it (by a committed transaction). Initially,  $committed = \emptyset$ .

The combined implementation of **ee** memory and  $Clients(n)$  is the module **Iee** whose possible actions are similar to that of **Iel**, but for  $t_1$ ,  $t_5$  and  $t_7$ , that are now:

- $t_1$ . If  $toabort = i > 0$ , then
  - 1) set  $imp\_out$  to  $\blacktriangleright_i$ ;
  - 2) for every  $(i, x, v) \in committed$ , set  $imp\_mem[x]$  to  $v$  and remove  $(i, x, v)$  from  $committed$ ; set  $trans[i] = \Lambda$  and  $toabort = 0$ ;
- $t_5$ . Set  $imp\_out$  to  $W_i(x, v)$ , append it to  $trans[i]$ , add  $(i, x, imp\_mem[x])$  to  $committed$ , and set  $imp\_mem[x]$  to  $v$ , if  $trans[i] \neq \Lambda$ , and for every  $j \neq i$ ,  $W(x), R(x) \notin trans[j]$ ;
- $t_7$ . Set  $imp\_out$  to  $\blacktriangleright_i$ , reset  $trans[i]$  to  $\Lambda$  and remove from  $committed$  every  $(i, x, v)$ , if  $trans[i]$  is not empty;

Module **Iee** has the same justice requirement as its predecessors.

To prove that **Iee** satisfies the specifications of Section IV, we cannot use the same  $R$  used to verify  $TM_1$ ; rather, we look at the “rolled back” version of memory values, which can be determined by  $committed$ . Formally, for each memory address  $x \in \mathbb{N}$ , we define

$$rolled\_back[x] = \begin{cases} v & \text{for some } j, \\ & (j, x, v) \in committed \\ imp\_mem[x] & \text{otherwise} \end{cases}$$

For the memory  $imp\_mem$ ,  $rolled\_back(imp\_mem)$  is  $imp\_mem$  where every entry is replaced by its rollback entry. Then the relation  $R_{STM}$  is defined by:

$$R_{STM}: \quad \begin{array}{l} spec\_out = imp\_out \wedge \bigwedge_{i=1}^n (q[i] = trans[i]) \wedge \\ spec\_mem = rolled\_back(imp\_mem) \end{array}$$

## VII. VERIFICATION WITH TLC

We verified the correctness of all implementations above by the explicit-state model checker TLC, the input of which are  $TLA^+$  programs. See [10] for a thorough discussion of TLC and  $TLA^+$ . Based on the similarity between TLC and the FDS model, we verified that all the implementations above indeed implement our trivial specification of Section IV.

To verify that an implementation correctly implements its specification, one has to provide  $TLA^+$  modules for both specification and implementation, and a mapping associating each of the specification’s variables with an expression over the implementation’s variables. With these, TLC verifies that the mapping is a refinement mapping satisfying the premises of Rule ABS-MAP. (In fact, the rule TLC uses is somewhat different, but suffices for our needs.) Since TLC can handle only finite-state systems, all parameters – memory size, number of clients, bound on pending transactions, etc. – have to be bounded.

### A. Specification Module

The specification module is constructed from two submodules, *Spec* and *Driver*. Submodule *Spec* is the core of the specification and is uniform for all  $TM$  specifications. It is essentially the specification module of Section IV. *Driver* defines features that are unique to each transactional memory by means of a resolving predicate  $\mathcal{F}$ . *Driver* can only restrict the next state relation and cannot introduce transitions that are not defined in Section IV.

### B. Implementation Module

All implementations include a module *Imp* that consists of a synchronous composition of the memory and the clients, such that every request by a client is immediately responded by the memory.

Since TLC requires that every *Spec* variable has a matching expression over *Imp* variables, we added a new variable to *Imp*,  $history\_q$ , which is a queue over  $E_{\perp}$  that contains all events of pending transactions. New events are appended to  $history\_q$ , and the events of a transaction that is closed (committed or aborted) are removed from it.

### C. Refinement mapping

The implementation module includes a mapping between *Spec*’s variables,  $spec\_mem$ ,  $q$ ,  $spec\_out$ , and  $doomed$ , to expressions over *Imp*’s variables. In all but our last example the refinement mapping is trivial:  $spec\_mem = imp\_mem$ ,  $q = history\_q$ ,  $spec\_out = imp\_out$ , and  $doomed[i] = F$  for all  $i$ . In the last example,  $spec\_mem = rolled\_back(imp\_mem)$  replaces  $spec\_mem = imp\_mem$ . TLC (automatically) verifies that the proposed mapping is a refinement mapping. Success means that, for the bounded instantiation taken, *Imp* implements its specification *Spec*, i.e., that every *Imp* implements some *Spec* run, and that every fair *Imp* computation maps into a fair *Spec* computation. In the first case, failure is indicated by a finite execution path leading from an initial state into a state in which the mapping is falsified. In the second case, failure is indicated by a finite execution path leading from an initial state to a loop in which the implementation meets all fairness requirements, and the associated specification does not.

## VIII. CONCLUSION AND FUTURE WORK

In this paper we developed a formal specification of transactional memory correctness and a methodology for verifying transactional memory implementations based on model checking. We demonstrated our approach on three transactional memory implementations drawn from the literature. While our models capture the important algorithmic aspects of those implementations, they are still quite a bit more abstract than “real” implementations in the form of C++ or Java libraries, say. The most obvious next step is to formally analyze more detailed models of implementations.

Practical transactional memory implementations must deal with memory accesses that occur outside of transactions. Such non-transactional accesses give rise to anomalies like the *privatization problem* [11], in which a thread can observe inconsistencies in what should be its own private copy of some shared data; and the *granular lost update problem* [12], in which the transactional implementation manages memory at a coarser granularity than changes made by nontransactional updates, leading to nontransactional updates being lost. It would be interesting to extend our formal specification and verification framework to account for non-transactional accesses and give precise and abstract characterizations of the privatization and GLU problems.

There are also a number of open questions concerning the programmer’s view of transactions, and we want to extend our framework to reason about them, too. For example,

- What happens when transactions contain other transactions? Two kinds of transaction nesting have been proposed: in *closed nesting* the nested transactions are “flattened” into one top-level transaction whose effects are invisible until commit time, while in *open nesting* the effects of nested transactions may be visible before commit. In open nesting the requirement for serializability is relaxed, and it would be interesting to extend our specification to account for this.

- What are the properties of various linguistic constructs for programming with transactions? This is an active area of research in the programming languages community (see [13] for one example).

Finally, we would like to harness the power of new verification technology like satisfiability modulo theories (SMT) that has already shown so much potential for software verification. Interesting questions are whether SMT and other software verification technology gives us additional leverage for efficient reasoning about transactional memory, and whether there are theories and decision procedures specific to transactional memory that we could add to the SMT arsenal.

## REFERENCES

- [1] M. Herlihy and J. E. B. Moss, “Transactional memory: architectural support for lock-free data structures,” in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 1993, pp. 289–300.
- [2] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool Publishers, 2007.
- [3] M. Scott, “Sequential specification of transactional memory semantics,” in *Proc. TRANACT the First ACM SIGPLAN Workshop on Languages, Compiler, and Hardware Support for Transactional Computing*, Ottawa, 2006.
- [4] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck, “Network invariants in action,” in *13th International Conference on Concurrency Theory (CONCUR02)*, ser. Lect. Notes in Comp. Sci., vol. 2421. Springer-Verlag, 2002, pp. 101–115.
- [5] M. Abadi and L. Lamport, “The existence of refinement mappings,” *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.
- [6] L. Hammond, W. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Herzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional memory coherence and consistency,” in *Proc. 31st annu. Int. Symp. on Computer Architecture*, Jun. 2004.
- [7] C. S. Ananian, K. Asanovic, B. C. Kuzmaul, C. E. Leiserson, and S. Lie, “Unbounded transactional memory,” in *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 316–327.
- [8] N. Shavit and D. Touitou, “Software transactional memory,” in *Proc. 14th ACM Symp. Princ. of Dist. Comp.* Ottawa Ontario CA: ACM Press, 1995, pp. 204–213.
- [9] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “Logtm: Log-based transactional memory,” in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006, pp. 254–265.
- [10] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [11] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott, “Privatization techniques for software transactional memory,” Department of Computer Science, University of Rochester, Tech. Rep. 915, Feb. 2007.
- [12] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. F. Moore, and B. Saha, “Enforcing isolation and ordering in STM,” in *ACM Conference on Programming Language Design and Implementation*, Jun. 2007.
- [13] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy, “Composable memory transactions,” in *PPoPP*, Jun. 2005.
- [14] Y. Kesten and A. Pnueli, “Control and data abstractions: The cornerstones of practical formal verification,” *Software Tools for Technology Transfer*, vol. 4, no. 2, pp. 328–342, 2000.
- [15] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag, 1995.

## APPENDIX

Fair Discrete Systems and Their Computations As a computational model for reactive systems we take the model of *fair discrete systems* (FDS) [14], which is a slight variation on the model of *fair transition system* [15]. Under this model, a system  $\mathcal{D} : \langle V, \mathcal{O}, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$  consists of the following components:

- $V$  — A set of *system variables*. A *state* of  $\mathcal{D}$  provides a consistent interpretation of the variables  $V$ . For a state

system variable  $v \in V$ , we denote by  $s[v]$  the value assigned to  $v$  by the state  $s$ . Let  $\Sigma$  denote the set of all states over  $V$ .

- $\mathcal{O} \subseteq V$  — A subset of *observable variables*. These are the variables which can be externally observed.
- $\Theta$  — The *initial condition*: An assertion (state formula) characterizing the initial states.
- $\rho(V, V')$  — The *transition relation*: An assertion, relating the values  $V$  of the variables in state  $s \in \Sigma$  to the values  $V'$  in an  $\mathcal{D}$ -successor state  $s' \in \Sigma$ . We assume that every state has a  $\rho$ -successor.
- $\mathcal{J}$  — A set of *justice (weak fairness)* requirements (assertions); A computation must include infinitely many states satisfying each of the justice requirements.
- $\mathcal{C}$  — A set of *compassion (strong fairness)* requirements: Each compassion requirement is a pair  $\langle p, q \rangle$  of state assertions; A computation should include either only finitely many  $p$ -states, or infinitely many  $q$ -states.

For an assertion  $\psi$ , we say that  $s \in \Sigma$  is a  $\psi$ -state if  $s \models \psi$ .

A *run* of an FDS  $\mathcal{D}$  is a possibly infinite sequence of states  $\sigma : s_0, s_1, \dots$  satisfying the requirements:

- *Initiality* —  $s_0$  is initial, i.e.,  $s_0 \models \Theta$ .
- *Consecution* — For each  $\ell = 0, 1, \dots$ , the state  $s_{\ell+1}$  is an  $\mathcal{D}$ -successor of  $s_\ell$ . That is,  $\langle s_\ell, s_{\ell+1} \rangle \models \rho(V, V')$  where, for each  $v \in V$ , we interpret  $v$  as  $s_\ell[v]$  and  $v'$  as  $s_{\ell+1}[v]$ .

A *computation* of  $\mathcal{D}$  is an infinite run that satisfies

- *Justice* — for every  $J \in \mathcal{J}$ ,  $\sigma$  contains infinitely many occurrences of  $J$ -states.
- *Compassion* — for every  $\langle p, q \rangle \in \mathcal{C}$ , either  $\sigma$  contains only finitely many occurrences of  $p$ -states, or  $\sigma$  contains infinitely many occurrences of  $q$ -states.

A *synchronous parallel composition* of systems  $\mathcal{D}_1$  and  $\mathcal{D}_2$ , denoted by  $\mathcal{D}_1 \parallel \mathcal{D}_2$ , is specified by the FDS

$$\mathcal{D} : \langle V_1 \cup V_2, \mathcal{O}_1 \cup \mathcal{O}_2, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2, \mathcal{J}_1 \cup \mathcal{J}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \rangle$$

To guarantee that the composition doesn't cause any computation of the composed system to be lost, we further require that for every  $i = 1, 2$ , each  $\mathcal{D}_i$ -computation is some computation of  $\mathcal{D}$  when projected onto  $V_i$ .

# Algorithmic Analysis of Piecewise FIFO Systems

Naghmeh Ghafari\*, Arie Gurfinkel<sup>†</sup>, Nils Klarlund<sup>‡</sup>, and Richard Trefler\*

\*David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

Email: nghafari@swen.uwaterloo.ca, trefler@cs.uwaterloo.ca

<sup>†</sup>Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA

Email: arie@sei.cmu.edu

<sup>‡</sup>Google Inc.

Email: klarlund@google.com

**Abstract**—Systems consisting of several components that communicate via unbounded perfect FIFO channels (i.e. FIFO systems) arise naturally in modeling distributed systems. Despite well-known difficulties in analyzing such systems, they are of significant interest as they can describe a wide range of communication protocols. Previous work has shown that piecewise languages play an important role in the study of FIFO systems.

In this paper, we present two algorithms for computing the set of reachable states of a FIFO system composed of piecewise components. The problem of computing the set of reachable states of such a system is closely related to calculating the set of all possible channel contents, i.e. the *limit language*. We present new algorithms for calculating the limit language of a system with a single communication channel and a class of multi-channel system in which messages are not passed around in cycles through different channels. We show that the worst case complexity of our algorithms for single-channel and important subclasses of multi-channel systems is exponential in the size of the initial content of the channels.

## I. INTRODUCTION

Concurrent systems consisting of a set of finite state machines that communicate via unbounded First-In First-Out (FIFO) channels are a common model of computation for describing distributed protocols such as IP-telecommunication protocols, interacting web services, and System on Chip (SoC) architectures (e.g., [9], [5], [1], [18], [10], [6], [20]). Even though all physically constructible systems have finite size channels, their size is often an implementation parameter that is typically left unspecified. Thus, modeling with unbounded channels is often more appropriate.

While unboundedness of communication channels provides a useful modeling abstraction, it complicates the analysis. Brand and Zafiropulo [9] showed that a single unbounded channel is already sufficient to simulate the tape of a Turing machine. Hence, verification of any non-trivial property, such as reachability, is undecidable. Despite these results, a substantial effort has gone into identifying subclasses of FIFO systems for which the verification problem is decidable (e.g., [1], [2], [3], [4], [5], [7], [8], [10], [16], [18]).

In this paper, we study the class of *piecewise* FIFO systems. These systems can be used for modeling distributed protocols such as IP-telecommunication protocols and interacting web services [16], [13]. A piecewise FIFO system is composed of components whose behaviors can be expressed by piecewise languages. Intuitively, a language is piecewise if it is accepted

by a non-deterministic finite state automaton whose only non-trivial strongly connected components are states with self-loops. Formally, a piecewise language is a union of sets of strings, where each set is given by a regular expression of the form  $M_0^* a_0 M_1^* \cdots a_{n-1} M_n^*$ , where each  $M_i$  is a subset of the alphabet  $\Sigma$  and each  $a_i$  is an element of  $\Sigma$ . In [16], [13], verification problems for piecewise FIFO systems were described and shown to be decidable.

Although piecewise languages may look restrictive, they can be used to express descriptions of IP-telephony features [16], [13] and seem amenable to describing composite web services specified in Business Process Execution Language (BPEL) [14]. For example, [13] studied the behavior of the telephony features in BoxOS which is the next generation telecommunication service over IP developed at AT&T Research [6], [15]. Essentially an active call is represented by a graph of telephony features (referred to as *boxes*) while communication between neighboring boxes is handled via unbounded perfect FIFO channels. At a sufficient level of abstraction, boxes may all be viewed as finite state transducers. It is required that the communication between different boxes follow a certain pattern. Thus, all of the feature boxes implement a communication template that consists of three phases (cf. [6]): *setup* phase, *transparent* phase, and *teardown* phase. Interestingly, as shown in [16], [13], this communication template can be expressed by piecewise languages.

It is crucial to be able to reason about safety and deadlock properties of BoxOS implementations with multiple features, something that the techniques in [6] fell short to address. It has been shown in [6], [16], [13] how aspects of BoxOS can be modeled as a piecewise FIFO system. In this paper, our main contribution is the algorithms for reachability analysis of piecewise FIFO systems. Analysis that was either not possible with previous approaches or for which the cost of the previous approaches was unknown.

The ability to calculate all possible channel contents that may arise from an initial state, i.e. the *limit language*, plays a central role for automated verification of non-trivial properties of FIFO systems. This problem is undecidable in general. Moreover, the limit language is not necessarily regular, even if the initial language is [10], and even when the limit language is known to be regular, determining it may still be undecidable [10]. Recently, in [16] it has been shown that

for the piecewise FIFO systems, the limit language is regular even when conditional actions are considered. However, the construction of the limit language may not always be effective.

We present two new algorithms for computing the limit language: one for single-channel systems, and another for multi-channel systems with acyclic communication graphs. In both algorithms, we use automata to represent and manipulate the set of possible channel configurations. We also discuss the complexity of these algorithms.

The algorithm for single-channel systems requires that components be piecewise, and applies to any regular initial channel content. We show that the worst case complexity of the algorithm is at most exponential in the size of the automaton that represents the language of the initial channel content.

The algorithm for the multi-channel systems requires that both the components and the initial contents of the channels be piecewise, and that the *communication graph* be acyclic. A communication graph is a graph with channels as vertices and conditional actions as edges indicating which channels are connected by these actions. For ease of presentation, we develop the algorithm incrementally by restricting the topology of the communication graphs to *star*, *tree*, *inverted tree*, and *directed acyclic graph (DAG)* topologies. We study the worst case complexity of the algorithm for each topology. We show that for the *star* and *tree* topologies the worst case complexity of the algorithm is exponential in the size of the automaton that represents the language of the initial content of the channel in the origin of the star and the root of the tree, respectively.

The rest of the paper is organized as follows. An overview of piecewise languages and their properties is given in Sec. II, and is followed by a description of the system model in Sec. III. The algorithm for single-channel systems is presented in Sec. IV, and the one for multi-channel systems in Sec. V. We review related work in Sec. VI, and conclude in Sec. VII.

## II. REGULAR AND PIECEWISE LANGUAGES

Let  $\Sigma$  be a finite alphabet and  $\epsilon$  the empty string. A regular expression (RE) over  $\Sigma$  is defined by the following grammar  $r ::= a \in \Sigma \mid r \cdot r \mid r + r \mid r^*$ . The language  $\mathcal{L}(r)$  denoted by an RE is defined in the usual way and we sometimes just write  $r$  to mean  $\mathcal{L}(r)$ . In a further abuse of notation, we often regard a set  $M \subseteq \Sigma \cup \{\epsilon\}$  as an RE, namely the sum of elements in  $M$ .

**Definition 1 (Piecewise Languages)** [16], [8] *A language is simply piecewise if it can be expressed by an RE of the form  $M_0^* a_0 \cdots a_{n-1} M_n^*$ , where each  $M_i \subseteq \Sigma$  and  $a_i \in \Sigma \cup \{\epsilon\}$ . A piecewise language is a finite (possibly empty) union of simply piecewise languages.*

For example,  $(a + b)^* c$  is simply piecewise, where  $M_0 = \{a, b\}$  and  $a_0 = c$ , but  $(ab)^* c$  is not piecewise.

**Definition 2 (FSA)** A finite state automaton (FSA)  $A$  is a tuple  $(\Sigma, Q, q^0, \delta, F)$ , where  $\Sigma$  is a finite alphabet;  $Q$  is a finite set of states;  $q^0 \in Q$  is the initial state;  $\delta : Q \times \Sigma \rightarrow 2^Q$  is the transition relation; and  $F \subseteq Q$  is a set of accepting (or final) states. When  $F$  is omitted, it is assumed that  $F = Q$ .

For  $a \in \Sigma$  we write  $\delta(q, a, q')$  to mean that  $q' \in \delta(q, a)$ . Given  $q \in Q$ , and  $w \in \Sigma^*$ ,  $\delta(q, w)$  is defined as usual:  $\delta(q, \epsilon) \triangleq \{q\}$ , and  $\delta(q, wa) \triangleq \{p \mid \exists r \in \delta(q, w), p \in \delta(r, a)\}$ . We say that a word  $w$  is accepted by  $A$  iff  $(\delta(q^0, w) \cap F) \neq \emptyset$ . The language of  $A$  is defined as  $\mathcal{L}(A) = \{w \in \Sigma^* \mid \delta(q^0, w) \cap F \neq \emptyset\}$ . We define the *size* of an FSA  $A$  as usual:  $|A| \triangleq |Q| + |\delta|$ .

We often use RE notation with automata. For example,  $A_1 \cdot A_2$  stands for concatenation of two automata,  $A_1 + A_2$  for an automaton with language  $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ , and  $(A_1 \text{ matches } a \cdot W)$  is true iff  $\mathcal{L}(A_1) = \mathcal{L}(a \cdot W)$ .

**Definition 3 (PO-FSA)** A partially ordered automaton (PO-FSA) is a tuple  $(A, \preceq)$ , where  $A = (\Sigma, Q, q^0, \delta, F)$  is an automaton, and  $\preceq \subseteq Q \times Q$  is a partial order on states such that  $q' \in \delta(q, a)$  implies that  $q \preceq q'$ .

**Proposition 1** [16] *A language is piecewise iff it is recognized by a PO-FSA.*

**Proposition 2** (cf. [16], [8]) *Piecewise languages are closed under finite unions ( $+$ ), finite intersections ( $\cap$ ), concatenation ( $\cdot$ ), shuffle ( $\parallel$ ), projections (defined by letter-to-letter mappings), and inverse homomorphisms, but not under complementation and substitutions.*

## III. SYSTEM MODEL

In this section, we review the definition of FIFO systems and the reachability problem for them.

**Action Languages and Semantics.** A channel over an alphabet  $\Sigma$  is a FIFO queue whose contents is given by a word  $w \in \Sigma^*$ . We define two types of channel actions: read  $a$ , denoted by  $?a$ , and write  $a$ , denoted by  $!a$ , that stand for reading and writing a letter  $a$  from/to a channel, respectively. We use  $f : w$  to denote the application of an action  $f$  to a word  $w$ . For example,  $?a : abb = bb$  and  $!a : bb = bba$ .

Let  $\Sigma_{rw} \triangleq \{?, !\} \times \Sigma$  denote read/write(rw)-alphabet over  $\Sigma$ . For a set of channels  $C = \{c_1, \dots, c_k\}$  this alphabet is extended as follows:  $\Sigma_{rw}(C) \triangleq [1..k] \times \Sigma_{rw}$ . Thus, an action  $4?a$  corresponds to reading  $a$  from channel  $c_4$ , and  $6!b$  corresponds to writing  $b$  to channel  $c_6$ . In the sequel, we drop  $C$  from the notation when it is clear from the context. We call  $\Sigma_{rw}$  an *action alphabet*, and any subset of  $\Sigma_{rw}^*$  an *action language*.

A *channel configuration* for a system with  $k$  channels is a  $k$ -tuple  $\mathbf{w} \in (\Sigma^*)^k$ . We use  $\langle w_1, \dots, w_k \rangle$  to denote a tuple, where  $w_i$  is the content of channel  $i$ . In single-channel systems, a configuration is just the content of the single channel. We use bold fonts to differentiate between channel configurations in multi-channel and single-channel systems. Let  $\mathbf{w}[i]$  denote the content of channel  $i$  in  $\mathbf{w}$  and  $\mathbf{w}[i \mapsto y]$  denote a channel configuration obtained from  $\mathbf{w}$  by replacing the content of channel  $i$  with  $y$ .

In the single-channel case, for  $X \subseteq \Sigma_{rw}^*$  and  $W \subseteq \Sigma^*$ , we use  $X : W$  to denote the result of applying a sequence of actions from  $X$  to a word in  $W$ . This is called the concrete semantics of actions and is defined as follows:

**Definition 4 (Concrete Semantics)** *Let  $W \subseteq \Sigma^*$  be a set of words over  $\Sigma$ , and  $X$  an action language, then  $X : W$  is*

defined as follows:

$$\begin{aligned} ?a : W &\triangleq \{u \mid a \cdot u \in W\} & !a : W &\triangleq \{w \cdot a \mid w \in W\} \\ \{x \cdot y\} : W &\triangleq y : (x : W) & X : W &\triangleq \bigcup_{x \in X} (x : W) \end{aligned}$$

For example,  $(\{?a!b, ?a!c\} : a) = \{b, c\}$ .

Def. 4 is extended to a  $k$ -channel system as follows. Given  $\mathbf{w} \in (\Sigma^*)^k$  and an action language  $X$ , then  $X : \mathbf{w}$  for a single action is defined as shown below,

$$i?a : \mathbf{w} \triangleq \mathbf{w}[i \mapsto (?a : \mathbf{w}[i])] \quad !a : \mathbf{w} \triangleq \mathbf{w}[i \mapsto (!a : \mathbf{w}[i])]$$

and is extended to words identically to Def. 4.

We write  $?a \rightarrow !b$  for a *conditional action* that means “ $b$  is written only if  $a$  is read first”. In other words,  $?a \rightarrow !b$  is an abbreviation for a sequence of simple actions:  $?a!b$ . Given an action alphabet  $\Sigma_{rwc}(C)$  over a set of channels  $C$ , we define a conditional action alphabet  $\Sigma_{rwc}(C)$  that treats conditional actions as letters:

$$\Sigma_{rwc}(C) \triangleq \Sigma_{rw}(C) \cup ((C \times \{?\} \times \Sigma) \cdot (C \times \{!\} \times \Sigma))$$

For a set of actions  $Act \subseteq \Sigma_{rwc}(C)$ , a *communication graph* of  $Act$ ,  $CG(Act)$ , is a digraph  $(C, E)$ , with an edge  $(i, j) \in E$  iff there are  $a$  and  $b$  in  $\Sigma$  such that  $i?a \rightarrow j!b$  is in  $Act$ .

**Definition 5 (FIFO System)** A *FIFO system* is a tuple  $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ , where  $\Sigma$  is a finite alphabet;  $C = \{c_1, \dots, c_k\}$  is a finite set of channels;  $Q$  is a finite set of control locations;  $q^0 \in Q$  is the initial control location; and  $\delta \subseteq Q \times \Sigma_{rwc} \times Q$  is a set of transition rules.

Note that in Def. 5, a FIFO system is defined with respect to a conditional action alphabet  $\Sigma_{rwc}$ . A *global state* of  $\mathcal{S}$  is a pair  $(q, \mathbf{w})$  where  $q$  is a state in  $Q$  and  $\mathbf{w}$  is a channel configuration. The transition relation  $\Delta$  of  $\mathcal{S}$  is a set of triples of the form  $((q, \mathbf{w}), op, (q', \mathbf{w}'))$ , where  $op \in \Sigma_{rwc}$ ,  $(q, op, q') \in \delta$ , and  $\mathbf{w}' \in (op : \mathbf{w})$ .

A FIFO system  $\mathcal{S}$  is *piecewise* if there exists a partial order  $\preceq$  on  $Q$  such that  $q' \in \delta(q, op)$  implies that  $q \preceq q'$ .

**FIFO Reachability Problem.** We are interested in the reachability problem: given a FIFO system  $\mathcal{S}$  and a set of initial configurations  $\mathbf{I}$ , what is the set of all reachable global states?

This problem can be reduced to computing the semantics (Def. 4) of a regular action language. That is, let  $\mathcal{S} = (\Sigma, C, Q, q^0, \delta)$ ,  $q \in Q$  some control location, and  $\mathbf{I}$  a set of initial configurations. Define a finite automaton  $A_q = (\Sigma_{rwc}, Q, q^0, \delta, \{q\})$  where  $q$  is the only accepting state. Then, the set of all reachable configurations of  $\mathcal{S}$  at control location  $q$  is  $(\mathcal{L}(A_q) : \mathbf{I})$ .

Finally, computing the semantics of a regular action language is itself reducible to the *limit language problem*: given a regular language of actions  $L_a$  and a regular language of channel content  $\mathbf{W}$ , compute the language of  $(L_a^* : \mathbf{W})$ . In the particular case of piecewise FIFO systems,  $L_a$  is further restricted to subsets of  $\Sigma_{rwc}$ . This is the problem we study in the rest of the paper.

```

1: Aut SINGLELIMIT (Aut  $A_I$ , Set  $Act$ )
2:    $R := \epsilon$ ,  $F := A_I$ 
3:   while  $\mathcal{L}(F) \not\subseteq \mathcal{L}(R)$  do
4:      $R := R + F$ 
5:      $F := \text{FULL}(F, Act)$ 
6:   end while
7:   return PARTIAL( $R, Act$ )

```

Fig. 1. The SINGLELIMIT algorithm.

#### IV. ANALYSIS OF SINGLE-CHANNEL PIECEWISE SYSTEMS

In this section, we focus on the analysis of a single-channel piecewise FIFO system. We present an algorithm for calculating the limit language, show its correctness, and discuss its worst case complexity.

Fig. 1 shows the algorithm SINGLELIMIT for calculating the limit language. The inputs to the algorithm are an automaton  $A_I$  representing a set of initial single-channel configurations  $I \subseteq \Sigma^*$ , and a set  $Act \subseteq \Sigma_{rwc}$  of actions; the output is an automaton representing the limit language  $(Act^* : I)$ .

The algorithm has two phases. In the first phase, called FULL, the algorithm iteratively computes all configurations reachable by (i) reading the current channel content completely, and (ii) writing the result of conditional and other write actions. Let  $Act \subseteq \Sigma_{rwc}$  be partitioned into unconditional write actions  $Act_w = \{!a \mid a \in Act\}$ , and the rest  $Act_r = Act \setminus Act_w$ . In each iteration, if  $W$  is the set of currently reachable configurations, the algorithm computes  $W' = \text{FULL}(W, Act)$  such that

$$W' = \{w \mid \exists u \in W, w \in (Act_r^{u|} : u)\} \cup (Act_w^* : \epsilon)$$

Note that FULL misses some reachable configurations. For example, let  $Act \triangleq \{?a \rightarrow !c, ?b \rightarrow !d, !e\}$  and  $I \triangleq ab$ . Then, FULL results in  $\mathcal{L}(e^*ce^*de^*)$  and misses reachable configurations in  $\mathcal{L}(be^*ce^*)$ . This is fixed in the second phase, called PARTIAL. Let  $W$  be a set of reachable configurations, the result of PARTIAL is a set  $W'$  such that

$$W' = \{w \mid \exists u, v, z, (v \cdot u \in W) \wedge (u \cdot z = w) \wedge (z \in \text{FULL}(\{v\}, Act))\}$$

These two phases are implemented using automata as described below.

**FULL Phase.** As inputs, FULL( $A, Act$ ) takes an automaton  $A = (\Sigma, Q, \delta, q^0, F)$ , and a set of actions  $Act$ . As output, it constructs an automaton  $A' = (\Sigma, Q, \delta', q^0, F)$ , where  $\delta'$  is defined as follows:

$$\begin{aligned} \delta'(q, i, q') \Leftrightarrow & (i = \epsilon \wedge \delta(q, a, q') \wedge \exists (?a) \in Act) \vee \\ & (i = b \wedge \delta(q, a, q') \wedge \exists (?a \rightarrow !b) \in Act) \vee \\ & (i = c \wedge q = q' \wedge \exists (!c) \in Act) \end{aligned}$$

Intuitively, the first rule of  $\delta'$  corresponds to unconditional reads (replacing by  $\epsilon$  transitions), the second – to renaming the labels of the transitions according to the conditional actions, and the third – to unconditional writes.

**PARTIAL Phase.** Let  $A = (\Sigma, Q, q^0, \delta, F)$  be an automaton and  $s$  be a state in  $Q$ . We construct two automata:  $A_1 = (\Sigma, Q, q^0, \delta, \{s\})$  and  $A_2 = (\Sigma, Q, \{s\}, \delta, F)$ . Let  $A'_1$  be the automaton constructed by applying FULL to  $A_1$ ,

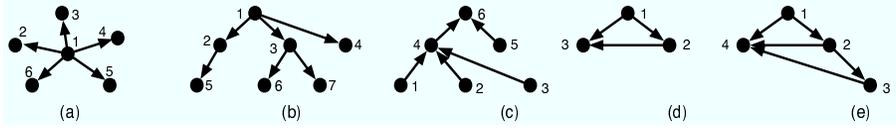


Fig. 2. Communication topologies: (a) star, (b) tree, (c) inverted tree, (d), (e) DAGs.

i.e.,  $A'_1 = \text{FULL}(A_1, \text{Act})$ . Then, the language of  $A_2 \cdot A'_1$  contains a word  $u \cdot z$  iff (i) there exists a word  $v$  such that  $v \cdot u$  is accepted by  $A$  via a run passing through the state  $s$ , and (ii)  $z \in \text{FULL}(\{v\}, \text{Act})$ . We call this operation  $\text{PREFIX}(A, s, \text{Act})$ . It is easy to see that:

$$\text{PARTIAL}(A, \text{Act}) = \bigcup_{s \in Q} \text{PREFIX}(A, s, \text{Act})$$

The algorithm in Fig. 1 always terminates. Given an automaton  $A$ ,  $\text{FULL}$  produces an automaton with the same number of states as  $A$ . Thus, the set  $\{\text{FULL}^i(A, \text{Act})\}_i$  is finite, and the algorithm always reaches a fixpoint.

**Theorem 1** *Let  $A_I$  be an automaton representing a set of configurations,  $\text{Act}$  be a set of actions, and  $A_L$  be the automaton returned by  $\text{SINGLELIMIT}(A_I, \text{Act})$ . Then,  $\mathcal{L}(A_L) = (\text{Act}^* : \mathcal{L}(A_I))$ .*

**Complexity Analysis.** Let  $h = |A_I|$  denote the size of  $A_I$  – the automaton representing the set of initial configurations. As discussed above,  $\text{FULL}(A_I, \text{Act})$  produces an automaton with the same number of states as  $A_I$  by relabeling the transitions of  $A_I$ . In the worst case, each transition can be updated at most  $|\Sigma|$  times. Thus, the worst case complexity of the  $\text{SINGLELIMIT}$  algorithm is  $|\Sigma|^h$ .

**Theorem 2** *Let  $A_I$  be an automaton over a finite alphabet  $\Sigma$  representing a set of single-channel configurations, and  $h = |A_I|$ . Then, in the worst case, the running time of the  $\text{SINGLELIMIT}$  algorithm is  $O(|\Sigma|^h)$ .*

## V. ANALYSIS OF MULTI-CHANNEL PIECEWISE SYSTEMS

In this section, we focus on the limit language problem for a set of actions,  $\text{Act}$ , on a  $k$ -channel system with an acyclic communication graph,  $\text{CG}(\text{Act})$ . For ease of presentation, we develop the algorithm for the DAG topology incrementally by restricting  $\text{CG}(\text{Act})$  to *star*, *tree*, *inverted tree*, and eventually *DAG* topologies. We show correctness of each algorithm and discuss its complexity.

Throughout, we assume that all actions are conditionals. This is not a significant limitation since: (i) unconditional reads can be modeled by conditionals that write to dummy channels, and (ii) unconditional writes can be handled easily, but are ignored for presentational convenience. The algorithms are based on automata, and operate on *piecewise configurations*. A piecewise ( $k$ -channel) configuration  $\mathbf{u}$  is a tuple  $\langle A_1, \dots, A_k \rangle$ , where each  $A_i$  is a PO-FSA over  $\Sigma$ . A piecewise configuration  $\mathbf{u} = \langle A_1, \dots, A_k \rangle$  represents a set of channel configurations  $\mathcal{L}(A_1) \times \dots \times \mathcal{L}(A_k)$ , which is denoted by  $\mathcal{L}(\mathbf{u})$ . The *size* of  $\mathbf{u}$  is the sum of the sizes of all of the automata in it.  $\mathcal{L}$  is extended to finite sets of piecewise configurations in the usual way:  $\mathcal{L}(\mathbf{U}) = \bigcup_{\mathbf{u} \in \mathbf{U}} \mathcal{L}(\mathbf{u})$ . Note that  $\mathcal{L}(\mathbf{U})$  can be seen as a piecewise recognizable relation.

For notational convenience, in the examples we use tuples of regular expressions instead of PO-FSA to represent piecewise configurations. For example,  $\mathbf{u} = \langle a^*b, (c+d)^*e \rangle$  represents a piecewise configuration where  $\mathbf{u}[1]$  is an automaton representing  $a^*b$ , and  $\mathbf{u}[2]$  is an automaton representing  $(c+d)^*e$ . In pseudo-code, we use **Conf** for the type of piecewise configurations, and notation  $X$  **with**  $[i] = y$  to mean  $X[i \mapsto y]$ .

### A. Star Topology

A set of actions  $\text{Act}$  has a *star* topology iff there exists a unique channel  $o$ , the *origin*, s.t. for every action  $i?a \rightarrow j!b$  in  $\text{Act}$ ,  $i = o$  and  $j \neq o$ , i.e.,  $\text{CG}(\text{Act})$  is a star (see Fig. 2(a)). In the sequel, we assume that channel 1 is the origin channel.

Let  $\mathbf{u}$  be a piecewise channel configuration. The algorithm  $\text{DOREAD}$ , shown in Fig. 4, computes the limit  $(\text{Act}^* : \mathcal{L}(\mathbf{u}))$ . The use of the argument *idx* is explained in Sec. V-D.  $\text{DOREAD}$  is driven by the automaton  $\mathbf{u}[1]$  representing the content of channel 1. For example, if  $\mathbf{u}[1] = M_1^*a_1M_2^*a_2$  then the algorithm first computes all reachable configurations  $\mathbf{w}$  whose channel 1 content,  $\mathbf{w}[1]$ , is in  $\mathcal{L}(M_1^*a_1M_2^*a_2)$ , then all configurations with  $\mathbf{w}[1]$  in  $\mathcal{L}(M_2^*a_2)$ , then all configurations with  $\mathbf{w}[1]$  being  $\epsilon$ . Each iteration of the algorithm is done using functions  $\text{SATURATE}$  and  $\text{STEP}$ . For our running example, in the first iteration,  $\text{SATURATE}$  computes all reachable configurations with  $\mathbf{w}[1]$  in  $\mathcal{L}(M_1^*a_1M_2^*a_2)$  and  $\text{STEP}$  computes all configurations with  $\mathbf{w}[1]$  in  $\mathcal{L}(M_2^*a_2)$ , etc.

$\text{SATURATE}$ . Let  $\mathbf{u}$  be a piecewise configuration, where  $\mathbf{u}[1]$  matches  $M^* \cdot Z$ , i.e.,  $\mathbf{u}[1]$  is a PO-FSA with a single initial state  $q^0$  and some self-loops on  $q^0$ . Note that,  $\mathbf{u}$  represents a set of configurations with an arbitrary number of letters from  $M$  at the head of channel 1. The  $\text{SATURATE}$  phase computes a set of configurations that are reachable by reading an arbitrary number of these letters. Formally,  $\text{SATURATE}(\mathbf{u}, 1)$  defines a piecewise configuration  $\mathbf{u}'$  such that  $\mathcal{L}(\mathbf{u}')$  is  $\{\mathbf{w} \mid \mathbf{w} \in (\text{Act}^* : \mathcal{L}(\mathbf{u})) \wedge (\mathbf{w}[1] \in \mathcal{L}(\mathbf{u}[1]))\}$ . It corresponds to a transformation  $\text{SATURATE}(\mathbf{u}, 1) = \mathbf{u}'$  such that

$$\mathbf{u}'[i] = \begin{cases} \mathbf{u}[1] & \text{if } i = 1 \\ \mathbf{u}[i] \cdot (\text{Act} : M)^* & \text{otherwise.} \end{cases}$$

$\text{STEP}$ . Let  $\mathbf{u}$  be a piecewise channel configuration, where  $\mathbf{u}[1]$  matches  $(a_0 + \dots + a_n) \cdot Z$ , i.e.,  $\mathbf{u}[1]$  is a PO-FSA with a single initial state with no self-loops. Here,  $\mathbf{u}$  represents a set of configurations whose channel 1 content starts with a letter in  $\{a_0, \dots, a_n\}$ . The  $\text{STEP}$  phase computes all configurations that are reachable by reading exactly one letter from channel 1. Formally,  $\text{STEP}(\mathbf{u}, 1)$  defines a set  $\mathbf{U}'$  of piecewise configurations such that  $\mathcal{L}(\mathbf{U}')$  is  $\{\mathbf{w} \mid \mathbf{w} \in (\text{Act}^* : \mathcal{L}(\mathbf{u})) \wedge (\mathbf{w}[1] \in Z)\}$ . It corresponds to a transformation

$$\text{STEP}(\mathbf{u}, 1) \triangleq \bigcup_{\{1?a \rightarrow i!b \in \text{Act} \mid a \in \{a_0, \dots, a_n\}\}} (1?a \rightarrow i!b) : \mathcal{L}(\mathbf{u})$$

```

1: Conf SATURATE(Conf u, Channel ch, int idx)
2: let  $(Q, q^0, \delta, F) = u[\text{ch}]$ ,  $M = \{a \mid (q^0, a, q^0) \in \delta\}$ 
3: forall  $i \in ([1..k] \setminus \{\text{ch}\})$  do
4:   let  $M' = \{b \mid (\text{ch}?a \rightarrow !b) \in \text{Act} \wedge a \in M \wedge$ 
       $(\forall j < \text{ch}, \text{idx}(j, b) = \text{idx}(j, a)) \wedge \text{idx}(\text{ch}, b) = \text{idx}\}$ 
5:    $u'[i] := (u[i] \cdot (M')^*)$ 
6:   return  $u'$ 
7: Set(Conf) STEP(Conf u, Channel ch, int idx)
8:  $U' := \emptyset$ 
9: let  $(Q, q^0, \delta, F) = u[\text{ch}]$ ,  $M = \{a \in \Sigma \mid \exists q', (q^0, a, q') \in \delta \wedge q' \neq q^0\}$ 
10: forall  $a \in M \wedge i \in \{j \mid \exists b, (\text{ch}?a \rightarrow !b) \in \text{Act}\}$  do
11:    $M' = \{b \mid (\text{ch}?a \rightarrow !b) \in \text{Act} \wedge$ 
       $(\forall j < \text{ch}, \text{idx}(j, b) = \text{idx}(j, a)) \wedge \text{idx}(\text{ch}, b) = \text{idx}\}$ 
12:    $u'[\text{ch}] := (Q, \delta(q^0, a) \setminus \{q^0\}, \delta, F)$ 
13:    $u'[i] := u[i] \cdot M'$ 
14:    $U' := U' \cup \{u'\}$ 
15: return  $U'$ 

```

Fig. 3. STEP and SATURATE algorithms.

```

1: global List readWL
2: global List writeWL
3: List DOREAD(Conf u, Channel ch)
4: doReadRec(u, ch, 0, true)
5: return writeWL
6: proc doReadRec(Conf u, Channel ch, int idx, bool na)
7: if  $u[\text{ch}]$  matches  $(U_1 + \dots + U_n)$  then
8:   for  $i \in [1..n]$  do doReadRec(u with  $[\text{ch}] := U_i$ , ch, idx, na)
9: else if  $u[\text{ch}]$  matches  $M^* \cdot W$  then
10:    $u := \text{SATURATE}(u, \text{ch}, \text{idx})$ 
11:    $\text{writeWL} = \text{writeWL} \cup \{u\}$ 
12:    $u := (u \text{ with } [\text{ch}] := W)$ 
13:   doReadRec(u, ch, idx + 1, false)
14: else if  $na \wedge u[\text{ch}]$  matches  $\epsilon$  then
15:    $\text{writeWL} := \text{writeWL} \cup \{u\}$ 
16: else if  $u[\text{ch}]$  matches  $a \cdot W$  then
17:   if  $na$  then  $\text{writeWL} := \text{writeWL} \cup \{u\}$ 
18:    $U := \text{STEP}(u, \text{ch}, \text{idx})$ 
19:   forall  $u' \in U$  do doReadRec( $u'$ , ch, idx + 1, true)
20: end if

```

Fig. 4. DOREAD algorithm for star topology and its supporting routines.

Detailed implementations of SATURATE and STEP for a set of actions  $Act$  on  $k$  channels are shown in Fig. 3.

**Theorem 3** *Let  $u$  be a piecewise channel configuration,  $Act$  an action set with star topology and origin  $o$ , and  $U'$  the set returned by  $\text{DOREAD}(u, o)$ . Then,  $\mathcal{L}(U') = (Act^* : \mathcal{L}(u))$ .*

**Complexity Analysis.** For a piecewise configuration  $u$ , the depth of the recursion of DOREAD is bounded by  $h = |u[o]|$  for the origin  $o$ . Inside each call, SATURATE takes constant time and returns a single configuration; however, STEP may return a set of configurations. In a  $k$ -channel system, the size of this set is bounded by  $k - 1$ . For example, for  $Act \triangleq \{1?a \rightarrow 2!b, 1?a \rightarrow 3!b\}$  and initial piecewise configuration  $\langle a, \epsilon, \epsilon \rangle$ , STEP returns two piecewise configurations  $\langle \epsilon, b, \epsilon \rangle$  and  $\langle \epsilon, \epsilon, b \rangle$ . Thus, the complexity of the DOREAD algorithm is bounded by the number of internal nodes of a  $(k - 1)$ -ary tree of height  $h$ . There are  $h$  such nodes for  $k = 2$ , and  $((k - 1)^{(h+1)} - 1)/(k - 2)$  for  $k > 2$ .

**Theorem 4** *Let  $u$  be a piecewise channel configuration,  $Act$  a set of actions with star topology on  $k$  channels with origin  $o$ , and  $h = |u[o]|$ . Then, in the worst case, the running time of  $\text{DOREAD}(u, o)$  is  $O(\max(k^h, h))$ .*

```

1: List TREELIMIT ()
2: for  $ch = 0$  to  $k$  do
3:    $\text{writeWL} := \emptyset$ 
4:   forall  $u \in \text{readWL}$  do doRead(u, ch)
5:    $\text{readWL} := \text{writeWL}$ 
6: return readWL

```

Fig. 5. The TREELIMIT algorithm.

```

1: Conf MERGES(Conf u, Channel ch);
2: Conf doWrite(Conf conf, Channel ch)
3: return MERGES(u, ch)
4: List MULTILIMIT ()
5: for  $ch = 0$  to  $k$  do
6:    $\text{writeWL} := \emptyset$ 
7:   forall  $u \in \text{readWL}$  do doRead(u, ch)
8:   if  $ch + 1 < k$  then
9:     forall  $u \in \text{writeWL}$  do
10:        $\text{readWL} := \text{readWL} \cup \{\text{doWrite}(u, ch + 1)\}$ 
11: return readWL

```

Fig. 6. MULTILIMIT algorithm and its supporting routines.

## B. Tree Topology

A set of actions  $Act$  has a *tree* (or, more generally, a *forest*) topology iff for all actions  $i?a \rightarrow j!b$  and  $i'a' \rightarrow j'!b'$  in  $Act$ ,  $j = j' \Rightarrow i = i'$ . That is,  $CG(Act)$  is a tree (e.g. Fig. 2(b)).

The DOREAD algorithm for the star topology is not applicable to the tree topology since it assumes that all reads come from a single channel. However, an action set with the tree topology can be partitioned such that each partition has a star topology. Formally, for a set of actions  $Act$ , let  $Act_i$  denote all the actions that read from channel  $i$ . Then,  $\{Act_i\}$  partitions  $Act$  and each  $Act_i$  has a star topology with origin  $i$ . For example, consider the CG in Fig. 2(b): here,  $Act_1 = \{1? \rightarrow 2!, 1? \rightarrow 3!, 1? \rightarrow 4!\}$ ,  $Act_2 = \{2? \rightarrow 5!\}$ , and  $Act_3 = \{3? \rightarrow 6!, 3? \rightarrow 7!\}$ .

This way, DOREAD can be used to compute  $Act_i^* : \mathcal{L}(u)$  for any channel  $i$  and a piecewise configuration  $u$ . Furthermore, it can be applied iteratively to compute sequential composition of the partitions of  $Act$ . For example, computation of  $(Act_1^* \cdot Act_2^*) : \mathcal{L}(u)$  is done by using DOREAD to first compute  $U'$  s.t.  $\mathcal{L}(U') = (Act_1^* : \mathcal{L}(u))$ , and using it again to compute  $(Act_2^* : \mathcal{L}(U'))$ . In the following, we show how to extend this to the computation of the full limit language.

The graph  $CG(Act)$  is acyclic and, therefore, induces a partial order  $\preceq$  on channels (vertices of the graph). For channels  $i$  and  $j$ ,  $i \preceq j$  iff there exists a path from  $i$  to  $j$  in  $CG(Act)$ . Intuitively, channel  $i$  is less than channel  $j$  if the final content of  $j$  depends on the initial content of  $i$ . We say that channel  $j$  *depends* on channel  $i$  if  $i \preceq j$ , and that  $i$  and  $j$  are *interdependent* if either  $i \preceq j$  or  $j \preceq i$ . W.l.o.g., we assume that the partial order  $\preceq$  is extended to a total order and that the channels are numbered such that  $i \preceq j$  iff  $i \leq j$ . For example, channel 3 depends on channels 1 and 2, and 2 depends only on 1. The ordering and renaming of the channels can be done in time linear in the size of the CG.

If  $Act$  has a tree topology, every channel in  $CG(Act)$  has at most one immediate predecessor. Thus, for every sequence  $x \in Act^*$ , there exists a sequence  $y$  such that: (i)  $y$  has the same actions as  $x$ , (ii) all reads of  $y$  are ordered, i.e.,  $y \in Act_1^* \cdot Act_2^* \cdot \dots$ , and (iii) if  $(x : w) \neq \emptyset$  for some  $w$ , then

$(y : \mathbf{w}) = (x : \mathbf{w})$ . For example, for  $Act$  in Fig. 2(b), and  $x = 1? \rightarrow 2! 1? \rightarrow 3! 2? \rightarrow 5! 1? \rightarrow 4! 3? \rightarrow 6!$ , an equivalent sequence  $y$  is

$$y = \underbrace{1? \rightarrow 2! 1? \rightarrow 3! 1? \rightarrow 4!}_{\in Act_1^*} \underbrace{2? \rightarrow 5!}_{\in Act_2^*} \underbrace{3? \rightarrow 6!}_{\in Act_3^*}$$

**Theorem 5** *Let  $Act$  be an action set on  $k$  channels s.t.  $CG(Act)$  is a tree, and  $\mathbf{w}$  a channel configuration. Then,*

$$((Act^* : \mathbf{w}) \neq \emptyset) \Rightarrow ((Act^* : \mathbf{w}) = ((Act_1^* \cdots Act_k^*) : \mathbf{w}))$$

Theorem 5 leads to an obvious algorithm for computing the limit language in the tree topology: (i) establish a total order on channels based on the CG, and (ii) use this order to iteratively apply DOREAD to each partition  $Act_i$ . We call this algorithm TREELIMIT (see Fig. 5). Since TREELIMIT proceeds through a finite total order of channels it always terminates.

**Theorem 6** *Let  $\mathbf{u}$  be a piecewise configuration,  $Act$  an action set with tree topology, and  $\mathbf{U}'$  the set of configurations returned by TREELIMIT. Then,  $\mathcal{L}(\mathbf{U}') = (Act^* : \mathcal{L}(\mathbf{u}))$ .*

**Complexity Analysis.** W.l.o.g., we assume that  $CG(Act)$  is an  $N$ -ary tree with  $M$  internal nodes and that the initial content of all the channels except the root is empty. Let  $\mathbf{u}$  be a piecewise configuration, and  $h = |\mathbf{u}|$ . By Theorem 4, computation of  $Act_i^* : \mathcal{L}(\mathbf{u})$  produces at most  $\max(N^h, h)$  piecewise configurations, each of size at most  $h$ . TREELIMIT applies computation  $Act_i^* : \mathcal{L}(\mathbf{u})$ ,  $M$  times, which produces at most  $\max(N^{h \times M}, h^M)$  configurations.

**Theorem 7** *Let  $\mathbf{u}$  be a piecewise configuration,  $Act$  a set of actions with a tree topology of degree  $N$  and  $M$  internal nodes, and  $h = |\mathbf{u}[1]|$ . In the worst case, the running time of TREELIMIT is  $O(\max(N^{h \times M}, h^M))$ .*

### C. Inverted Tree Topology

A set of actions  $Act$  has an inverted tree topology iff for all conditional actions  $i?a \rightarrow j!b$  and  $i'?a' \rightarrow j'!b'$  in  $Act$ ,  $i = i' \Rightarrow j = j'$ . That is,  $CG(Act)$  is an inverted tree (e.g., see Fig. 2(c)).

In the inverted tree topology, a channel may depend on several pairwise independent channels. Therefore, Theorem 5 is no longer applicable. For example, let  $Act \triangleq \{1?a \rightarrow 3!a, 2?b \rightarrow 3!b\}$ , and  $\mathbf{w} \triangleq \langle aa, bb, \epsilon \rangle$  be a configuration. The partial order on the channels induced by  $CG(Act)$  is  $\{1 \preceq 3, 2 \preceq 3\}$ , with two obvious linearizations. A configuration  $\langle \epsilon, \epsilon, abab \rangle$  is reachable from  $\mathbf{w}$ , but does not belong to neither  $((1?a \rightarrow 3!a)^*(2?b \rightarrow 3!b)^*) : \mathbf{w}$ , nor  $((2?b \rightarrow 3!b)^*(1?a \rightarrow 3!a)^*) : \mathbf{w}$ , which contradicts the theorem.

For simplicity of presentation, we assume that there is a unique channel, referred to as  $l$ , that has multiple dependencies, like channel 3 in the above example. That means  $l$  is the only channel whose node in  $CG(Act)$  has an in-degree greater than or equal to 2. In this case, it is possible to (i) replace channel  $l$  with new channels, called *shadows* of  $l$ , and turn  $Act$  into a tree topology, (ii) solve the new limit problem using TREELIMIT, and (iii) combine the contents of shadow channels together. This is further explained below.

We define a function ADDS that introduces shadow channels for  $l$  by redirecting each conditional that reads from  $i$  and writes to  $l$  to write to a newly created shadow channel  $\hat{l}_i$ . Formally,

$$\text{ADDS}(i?a \rightarrow j!b, l) \triangleq \begin{cases} i?a \rightarrow \hat{l}_i!b & \text{if } j = l \\ i?a \rightarrow j!b & \text{otherwise.} \end{cases}$$

ADDS breaks dependencies between channels. Let  $\widehat{Act} = \text{ADDS}(Act, l)$ . If  $CG(Act)$  is an inverted tree, then  $CG(\widehat{Act})$  is a tree. We use  $\mathbf{S}(l)$  to denote the shadows of  $l$ .

Let  $\mathbf{w}$  be a configuration, and  $\hat{\mathbf{w}}$  be its extension to shadow channels. That is,  $\hat{\mathbf{w}}[i] = \mathbf{w}[i]$  if  $i \notin \mathbf{S}(l)$ , and  $\hat{\mathbf{w}}[i] = \epsilon$  otherwise. The sets  $(Act^* : \mathbf{w})$  and  $(\widehat{Act}^* : \hat{\mathbf{w}})$  are closely related. Let  $\mathbf{t} \in (x : \mathbf{w})$  be a configuration reachable from  $\mathbf{w}$  by a sequence  $x \in Act^*$ , and  $\hat{\mathbf{t}} \in (\text{ADDS}(x, l) : \hat{\mathbf{w}})$  be a configuration reachable from  $\hat{\mathbf{w}}$ , where ADDS is extended to sequences in an obvious way. ADDS only augments actions that write to  $l$ . Thus,  $\mathbf{t}[i] = \hat{\mathbf{t}}[i]$  for any  $i$  that is different from  $l$  or its shadow channels  $\mathbf{S}(l)$ . By adding shadow channels for  $l$ , all the writes on  $l$  are redirected to its shadows and  $\hat{\mathbf{t}}[l]$  is the initial content of  $l$ , hence, it is a prefix of  $\mathbf{t}[l]$ . Each shadow channel  $\hat{l}_i$  keeps track of what was read from channel  $i$  and written to  $l$ , hence,  $\hat{\mathbf{t}}[\hat{l}_i]$  is a subsequence of  $\mathbf{t}[l]$ .

In order to formalize the relation between  $(Act^* : \mathbf{w})$  and  $(\widehat{Act}^* : \hat{\mathbf{w}})$ , we define a function MERGES. Given a configuration over shadow channels, MERGES produces all corresponding configurations without shadows. Formally,

$$\mathbf{t} \in \text{MERGES}(\hat{\mathbf{t}}, l) \Leftrightarrow (\forall i \neq l \wedge i \notin \mathbf{S}(l), \mathbf{t}[i] = \hat{\mathbf{t}}[i]) \wedge (\mathbf{t}[l] \in \mathcal{L}(\hat{\mathbf{t}}[l] \cdot \prod_{j \in \mathbf{S}(l)} \{\hat{\mathbf{t}}[j]\}))$$

**Theorem 8** *Let  $Act$ ,  $\widehat{Act}$ ,  $\mathbf{w}$ ,  $\hat{\mathbf{w}}$ , and  $l$  be as above. Then,*

$$\mathbf{t} \in (Act^* : \mathbf{w}) \Leftrightarrow \exists \hat{\mathbf{t}} \in (\widehat{Act}^* : \hat{\mathbf{w}}), \mathbf{t} \in \text{MERGES}(\hat{\mathbf{t}}, l)$$

Both Theorem 8 and MERGES are easily lifted to piecewise configurations s.t. if  $\mathbf{u}$  is a piecewise configuration, then  $\text{MERGES}(\mathbf{u}, l)$  defines a piecewise configuration as well. This follows from the fact that piecewise languages are closed under concatenation and shuffle (see Prop. 2).

The explained procedure can be extended to an arbitrary inverted tree. The correctness follows by induction on the number of channels. The final algorithm MULTILIMIT is shown in Fig. 6. The algorithm assumes that shadow channels are introduced where they are needed. It traverses the channels according to the partial order induced by the CG, applying read and write phases. The read phase is the same as in the star and tree topologies (done by DOREAD). The write phase uses MERGES to merge the content of all the shadows of a channel before applying a read phase to it.

**Theorem 9** *Let  $\mathbf{u}$  be a piecewise configuration,  $Act$  an action set with inverted tree topology, and  $\mathbf{U}$  the set of configurations returned by MULTILIMIT. Then,  $\mathcal{L}(\mathbf{U}) = (Act^* : \mathcal{L}(\mathbf{u}))$ .*

#### D. DAG Topology

In this section, we present an algorithm for computing the set of reachable configurations for a set of actions whose CG is an arbitrary directed acyclic graph (DAG) (e.g. see Fig. 2(d-e)). This subsumes the algorithms from the previous sections for star, tree, and inverted tree topologies.

What makes the DAG topology different from the inverted tree is that immediate predecessors (in the  $\preceq$  partial order on the CG) of a channel may be interdependent. For example, consider  $Act \triangleq \{1?a \rightarrow 3!a, 1?b \rightarrow 2!b, 2?b \rightarrow 3!b\}$  whose CG is shown in Fig. 2(d). Channel 3 has channels 1 and 2 as its immediate predecessors, and channel 2 depends on channel 1. This extra layer of dependence precludes the possibility of breaking the topology by simply introducing shadow channels.

For our running example, consider the computation of reachable configurations starting from  $\langle a^*b^*, \epsilon, \epsilon \rangle$ . We can replace channel 3 with two shadow channels to obtain  $\widehat{Act} = \{1?a \rightarrow \hat{3}_1!a, 1?b \rightarrow 2!b, 2?b \rightarrow \hat{3}_2!b\}$ . By applying TREELIMIT to the resulting tree topology, we obtain two piecewise configurations  $\{\langle a^*b^*, \epsilon, a^*, \epsilon \rangle, \langle b^*, b^*, a^*, b^* \rangle\}$ . If we then proceed by merging the contents of the shadows of channel 3, as in the inverted tree topology, we obtain  $\{\langle a^*b^*, \epsilon, a^* \rangle, \langle b^*, b^*, (a+b)^* \rangle\}$ . The second piecewise configuration includes configurations in which the content of channel 3 is in  $b^+a^+$ . These configurations are infeasible since  $a$  came before  $b$  in channel 1 in any initial configuration and this order must be preserved when the content is copied to channel 3.

To solve this problem, we extend MULTILIMIT algorithm by modifying the shuffle used by MERGES (see Sec. V-C) to respect the dependencies between the predecessors of the channel whose shadows are merged. This requires (i) keeping track of the relative positions of each letter in a channel as it is copied between channels, and (ii) restricting the shuffle based on the history of positions of each letter.

For a system with  $k$  channels, each letter is associated with a  $k$ -tuple of indices from  $IDX^k$ , where  $IDX \triangleq [-1..∞)$ . Intuitively,  $j$ th index of a letter  $a$  indicates the relative position of  $a$  when it was in channel  $j$ , with  $-1$  meaning that  $a$  was never in that channel. Let  $idx(i, a)$  be a function that extracts the  $i$ th index of  $a$ . For example,  $idx(2, a) = 4$  means that  $a$  was at some point at position 4 in channel 2, and  $idx(3, a) = -1$  means that  $a$  was never in channel 3. We use  $ch(a)$  to denote the latest channel that  $a$  was in. Formally,  $ch(a) \triangleq \max\{i \mid idx(i, a) \neq -1\}$ .

To keep track of the indices, several parts of the MULTILIMIT are modified. DOREAD (see Fig. 4) is extended to accept as an argument the  $ch$ -index of a letter at the head of the current channel  $ch$ , and increment it at each recursive call (lines 15 and 24). SATURATE and STEP (see Fig. 3) are extended to propagate and assign indices as well (lines 4 and 11).

The interdependence of the channels implies the following constraint on the content of every channel in every reachable configuration. Let  $w$  be a word describing a content of channel  $l$ . Let  $a$  and  $b$  be letters at positions  $p$  and  $q$  in  $w$ , respectively.

Assume that  $i$  is the last channel  $a$  was in, and that  $i$  precedes the last channel that  $b$  was in, i.e.,  $i = ch(a) < ch(b)$ . Furthermore, assume that  $a$  preceded  $b$  in channel  $i$ , i.e.,  $idx(i, a) < idx(i, b)$ . Then  $a$  has to precede  $b$  in  $w$ , i.e.,  $p < q$ , since  $a$  had to be read from channel  $i$  (and placed in  $w$ ) before  $b$  could be read.

We denote the set of all words that satisfy the above condition by WO. Formally, it is the set of all words  $w$  in  $(\Sigma \times IDX^k)^*$  that satisfy

$$\forall p, q, (a = w(p) \wedge b = w(q) \wedge i = ch(a) \wedge ch(a) < ch(b) \wedge idx(i, a) < idx(i, b)) \Rightarrow p < q$$

where  $w(p)$  denotes the letter at position  $p$  of  $w$ .

For our running example, the word  $ba$  in channel 3 does not belong to WO: the last channel of  $a$  is 1 which precedes 2 – the last channel of  $b$ , and  $a$  preceded  $b$  in channel 1, thus, it must also precede  $b$  in channel 3.

The set WO defines a piecewise language, and is recognizable by a PO-FSA.

**Theorem 10** *The language WO is piecewise.*

In order to restrict MERGES to only include words that satisfy WO, we replace it with a function MERGEDAGS defined as follows. Let  $\hat{t}$  be a configuration reachable from an initial configuration extended with shadow channels, and  $l$  a non-shadow channel. Then,  $MERGEDAGS(\hat{t}, l) \triangleq MERGES(\hat{t}, l) \cap WO$ . Since WO is piecewise (by Theorem 10) and piecewise languages are closed under intersection (by Prop. 2), MERGEDAGS defines a piecewise configuration.

With this change, MULTILIMIT algorithm computes the exact set of reachable configurations.

**Theorem 11** *Let  $\mathbf{u}$  be a piecewise configuration,  $Act$  a set of actions with DAG topology and  $\mathbf{U}'$  a set of configurations returned by MULTILIMIT algorithm, where MERGES is replaced by MERGEDAGS. Then  $\mathcal{L}(\mathbf{U}') = (Act^* : \mathcal{L}(\mathbf{u}))$ .*

## VI. RELATED WORK

FIFO systems play key roles in description and analysis of distributed systems. It is well-known that most non-trivial verification problems for FIFO systems are undecidable [9]. However, a substantial effort has gone into analysis of these systems. In general, two main approaches have been followed for the analysis of FIFO systems. The first approach, and the one taken in this paper, is to identify practically useful subclasses of FIFO systems with decidable properties (e.g., [18], [11], [19], [16]). The second approach is to look for efficient semi-algorithms that scale to realistic examples, but do not guarantee to always terminate (e.g., [4], [5], [12]). Although this approach may look promising, in many cases finding a good bound between scalability and termination is very challenging.

The two approaches may be combined, as illustrated in the analysis of lossy channel systems in which channels may lose messages. In these systems, the problem of reachability of a given state is decidable [2], [10], [1]; however, calculating the set of all reachable states is impossible. The systems

considered in this paper are not lossy; all channels are perfect, i.e., they do not lose any message.

Pachl [18] proves that if the set of reachable channel configurations (the limit language) is recognizable then it is decidable to check for reachability of any given state. It was later shown in [10] that even though the reachability set might be recognizable, determining it may still be undecidable.

Boigelot *et al.* [4], [5], [3] describe a data structure, QDD, for representing sets of queue contents, and a QDD-based semi-algorithm to compute a set of reachable states. The termination of this algorithm depends on handling iterations of arbitrary sequences of actions. This is equivalent to limit languages in our terminology. In [4], automata-theoretic algorithms are given to calculate  $f : L$  and  $f^* : L$  for a *single* read, write, or conditional action  $f$ . Boigelot's Ph.D. thesis [3] and [5] extend that to action sequences that preserve recognizability of channel contents. The key difference between [4], [5] and our work is that we concentrate on iteration of multiple conditional actions. Such sequences are harder to handle since in general they do not preserve recognizability of channel contents [16].

The problems of recognizability of limit languages and decidability of their computation for piecewise FIFO systems were first studied in [16]. Although the paper presents decidability results, it only sketches the algorithms and does not analyze their complexity. For the single-channel case, our new algorithm is simpler. It avoids the use of recurrence arguments on set-theoretic operators on alphabets. Rather, we bound the iterative construction through a direct use of the union operation on simpler versions of the original automata than the one presented in [16]. For the multi-channel case, we provide new explicit algorithms. Our concept of indexing allows dependencies between inputs on different channels to be added and tracked in the transducer-like treatment of the DAG topology.

An approach for model-checking piecewise FIFO systems was studied in [13]. That work presents a procedure for calculating an *abridged* model of a FIFO system, which when successful, constructs such a model by computing an abstraction of the reachable channel contents. It is shown in [13] that abridged models preserve path properties expressed by a restricted class of Büchi automata. In contrast, the work presented in this paper focuses on calculating the *exact* limit languages and applies to reachability/safety properties only.

## VII. CONCLUSION

FIFO systems are a common model of computation for distributed protocols. We have studied the reachability problem for a class of FIFO systems composed of piecewise components. Since it is well-known that this problem is reducible to computing the limit language of a regular language of actions, we concentrate exclusively on the limit language problem.

We consider single-channel and multi-channel FIFO systems separately. For the single-channel case, we present a new automata-theoretic algorithm for calculating the limit language starting with an arbitrary regular initial content. We show that the worst case complexity of our algorithm is exponential

in the size of the automaton representing the initial channel content. A prototype of the algorithm was implemented using the Automaton package [17].

For multi-channel systems, we present an automata-theoretic algorithm for computing the limit language subject to the following conditions: (i) the initial language is piecewise, and (ii) the communication graph of actions is acyclic. For the star and the tree topology, we show that the complexity of our algorithm is exponential in the size of the automaton representing the initial channel configuration. In the cases of inverted tree and DAG topologies the complexity of the algorithms remains an open problem.

**Acknowledgments.** Ghafari and Trefler are supported in part by grants from NSERC of Canada.

## REFERENCES

- [1] P. A. Abdulla, A. Annichini, and A. Bouajjani. "Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol". In *TACAS'99*, volume 1579 of *LNCS*, pages 208–222, 1999.
- [2] P. A. Abdulla and B. Jonsson. "Verifying Programs with Unreliable Channels". In *LICS'93*, pages 160–170, 1993.
- [3] B. Boigelot. *Symbolic Method for Exploring Infinite States Spaces*. PhD thesis, Université de Liège, 1998.
- [4] B. Boigelot and P. Godefroid. "Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs". *Formal Methods in System Design*, 14(3):237–255, 1999.
- [5] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. "The Power of QDDs". In *SAS'97*, volume 1302 of *LNCS*, pages 172–186, 1997.
- [6] G. W. Bond, F. Ivančić, N. Klarlund, and R. Trefler. "ECLIPSE Feature Logic Analysis". In *2nd IP Telephony Workshop*, 2001.
- [7] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. "Regular Model Checking". In *CAV'00*, volume 1855 of *LNCS*, pages 403–418, 2000.
- [8] A. Bouajjani, A. Muscholl, and T. Touili. "Permutation Rewriting and Algorithmic Verification". In *LICS'01*, pages 399–408, 2001.
- [9] D. Brand and P. Zafiropulo. "On Communicating Finite-State Machines". *Journal of the ACM*, 30(2):323–342, 1983.
- [10] G. Cece, A. Finkel, and S. P. Iyer. "Unreliable Channels are Easier to Verify than Perfect Channels". *Information and Computation*, 124(1):20–31, 1996.
- [11] A. Finkel and L. Rosier. "A Survey on the Decidability Questions for Classes of FIFO Nets". In *Advances in Petri Nets 1988*, volume 340 of *LNCS*, pages 106–132. Springer, 1988.
- [12] T. Le Gall, B. Jeannot, and T. Jérón. "Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues". In *AMAST'06*, volume 4019 of *LNCS*, pages 204–219, 2006.
- [13] N. Ghafari and R. J. Trefler. "Piecewise FIFO Channels Are Analyzable". In *VMCAI'06*, volume 3855 of *LNCS*, pages 252–266, 2006.
- [14] IBM. *Business Process Execution Language for Web Services (BPEL) Version 1.1*, 2007. Available from <http://www-128.ibm.com/developerworks/library/specification/ws-bpel>.
- [15] M. Jackson and P. Zave. "Distributed Feature Composition: A Virtual Architecture for Telecommunications Services". *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998.
- [16] N. Klarlund and R. J. Trefler. "Regularity Results for FIFO Channels". *Electronic Notes in Theoretical Computer Science*, 128(6):21–36, 2005.
- [17] A. Møller. "An Automaton/Regular Expression Library for Java". Available at <http://www.brics.dk/automaton>, 2007.
- [18] J. K. Pachl. "Protocol Description and Analysis Based on a State Transition Model with Channel Expressions". In *PSTV'87*, pages 207–219, 1987.
- [19] A. P. Sistla and L. D. Zuck. "Automatic Temporal Verification of Buffer Systems". In *CAV'91*, pages 59–69, 1991.
- [20] P. Wodey, G. Camaroque, F. Baray, R. Hersemeule, and J.-P. Cousin. "LOTOS Code Generation for Model Checking of STBus Based SoC: the STBus interconnect". In *MEMOCODE'03*, pages 204–216, 2003.

## Transaction Based Modeling and Verification of Hardware Protocols

Xiaofang Chen<sup>1</sup>, Steven M. German<sup>2</sup>, Ganesh Gopalakrishnan<sup>1</sup><sup>1</sup>School of Computing, University of Utah<sup>2</sup>IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

**Abstract**—Modeling hardware through atomic guard/action transitions with interleaving semantics is popular, owing to the conceptual clarity of modeling and verifying the high level behavior of hardware. In mapping such specifications into hardware, designers often decompose each specification transition into sequences of implementation transitions taking one clock cycle each. Some implementation transitions realizing a specification transition overlap. The implementation transitions realizing different specification transitions can also overlap.

We present a formal theory of refinement, showing how a collection of such implementation transitions can be shown to realize a specification. We present a modular refinement verification approach by developing abstraction and assume-guarantee principles that allow implementation transitions realizing a single specification transition to be situated in sufficiently general environments. Illustrated on a non-trivial VHDL cache coherence engine, our work may allow designers to design high performance controllers without being constrained by fixed automated synthesis scripts, and still conduct modular verification.

## I. INTRODUCTION

With the growing complexity of the internal organization of modern processors and other digital systems, it is important to develop notations, verification methodologies and tools that ensure correctness as well as high performance of the overall design, and allow designers to make design choices. To put things into sharper focus, consider the design of a modern high performance cache coherence protocol as an example. A designer initially conceptualizes the design in terms of atomic transitions that fire under certain conditions, and move request/response packets between various directories, caches, and processor cores. Accumulated experience, starting from Yang et al's early work on UltraSparc-1 [1] to more modern ones [2]–[4], shows that designs captured at this level of abstraction (*specification level*) in languages such as Murphi [5] and TLA+ [6] supporting guard/action style rules can be model checked to eliminate virtually all high level concurrency bugs. It is also widely known that the atomic transitions used at the specification level are implemented in hardware over multiple clock cycles (a *transaction* in our terminology), with one or more *implementation steps* happening in each clock cycle of the transaction. In addition, while the specification level models the desired computation according to the *interleaving* model where only one specification transition fires at a time, the implementation level often starts a second transaction before the first transaction has finished, again to maximize overlapped computation, pipelining, and to take advantage of internal buffers and split transaction buses. In today's design contexts, designers are seriously under-equipped concerning

(i) notations that allow them to specify the designs of such aggressively optimized implementations, (ii) theories for formally relating such implementations against specifications, and (iii) compositional methods for verifying implementations against specifications that have already been verified at the interleaving model level for global properties.

This paper makes a contribution in all these areas, surveying available solutions and contrasting our solutions against them, and reporting a complete case study in which a VHDL level cache coherence protocol engine is compositionally shown to implement a high level Murphi specification. Our overall approach is as follows. We employ an extended form of Murphi to model and verify specifications at the level of interleaving transitions for global properties (e.g., coherence) using model checking. We employ *HMurphi* (Hardware Murphi) to represent implementations. Hardware Murphi extends the Murphi language of Dill *et al* [5] into a hardware description language. S. German and G. Janssen defined Hardware Murphi and implemented a translator, called *muv*, from Hardware Murphi into synthesizable VHDL [7], [8]. Using the generated VHDL, we can apply symbolic model checking to either the interleaving specification or the implementation. HMurphi has the following features. First, it supports *signals* with modes *in*, *out*, and *internal*, and signal assignments. Second, in HMurphi, updates to variables within a rule are immediate, while writes become visible to all the other rules in the next step – thus modeling hardware at the level of clock cycles. Third, HMurphi follows the concurrent execution semantics (to be defined shortly), and directly supports transactions.

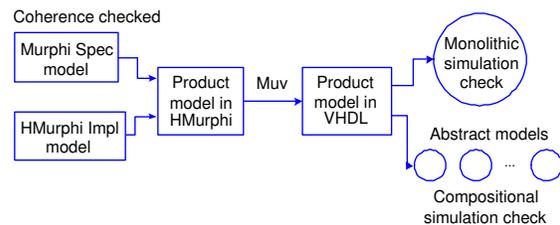


Fig. 1. The workflow of our approach

Given a coherence checked specification and an HMurphi implementation model (Figure 1), the theory described in this paper allows refinement between the implementation and the specification to be checked as assertions generated and embedded within a product model between the specification and implementation models. The product model can be checked (monolithically) using a VHDL model checker, or decomposed into a collection of abstract models, to perform a compositional proof of refinement. We have also designed

This work is supported in part by SRC Contract 2005-TJ-1318.

a VHDL version of the German cache coherence protocol [8], [9] – a widely used benchmark protocol in this area, and applied our verification methodology. Our experimental results are encouraging, and also suggest future research areas.

### A. Related Work

Direct modeling using finite-state machines expressed in VHDL/Verilog is unsuitable for representing, manipulating, and optimizing the control logic at the implementation level of concurrent transactions described above. Møllergaard *et al* [10] presented an approach to model and verify (using theorem proving) digital systems using Synchronized Transitions [11]. There was no attempt to model implementations separately, nor to verify correspondence. However, synchronized transitions, and similar notations including Unity [12], term rewriting systems [13], Murphi, TLA+, and Bluespec [14] reveal the wide adoption of the guard/action notation among designers.

Bluespec [13], [15], [16] is probably the most closely related work to our overall project. Both approaches view the specification in terms of interleaved executions. The Bluespec compiler automatically synthesizes the specification into a RTL model, automatic scheduling the implementation actions according to pre-defined synthesis recipes.

However, such automatic synthesis methods may not meet performance goals. They may also not give designers enough flexibility in handling pre-existing interfaces such as embedded split-transaction buses or packet routing networks around which implementations are designed. By allowing designers to express implementation details in terms of transactions and supporting the verification of refinement, our approach is better suited to existing industrial design flows.

Our notion of transactions is related to Park’s approach to aggregation of distributed actions [17]. Park’s work uses theorem proving while we develop a compositional approach using model checking. Also, our work is directed towards verifying hardware implementations, while Park verifies abstract protocols.

Our compositional verification approach relies on input variable generalization and assume-guarantee reasoning. Many previous approaches to assume-guarantee reasoning have been developed for both software and hardware contexts, for example, [18], [19]. Although we verify hardware, our model is related to models of concurrent software with interfering operations.

In our approach to abstraction, variables of the model can be abstracted to different degrees at different points in time. At some steps, a variable can be abstracted by input variables, while at other steps, the variable may not be abstracted. This selective abstraction approach has some resemblances with how the degree of input weakenings can be controlled in STE based verification (e.g., as in [20, Page 25]); however, virtually all details are different.

A goal of our research is to develop methods for checking refinement that are computationally efficient and also reduce the manual burden of specifying and verifying the mapping between the implementation and the specification. To reduce the manual effort, we develop a theory of refinement checking

that incorporates the notion of multi-cycle transactions and their mappings onto interleaving specifications. The kind of refinement checking we perform can be formulated in other theories, but we believe that in many cases the necessary mappings and reasoning will be simpler in our theory.

### B. Overview of the paper

(Reading suggestion: this section, then Sections VII and VIII, then II onwards.) Section II discusses notational preliminaries. We define transitions and their read and write sets of variables. A transition in the implementation is a single clock cycle operation, while a transaction is a multi-cycle operation. Implementation transactions are viewed as sets of transitions, with each set helping implement a specification transition.

Section III discusses the notion of a correct implementation of a specification.

Section IV discusses monolithic model checking by defining a product model between the specification and implementation models. The cross product model has pre- and postcondition on transitions to check that the implementation correctly implements the specification.

Section V discusses compositional model checking where a collection of abstract models are created through read-variable generalization and assume-guarantee reasoning.

Section VI discusses our heuristics to implement compositional model checking.

## II. SOME PRELIMINARIES

### A. States, Transitions, Models

Let  $V$  be a set of variables and  $D$  be a set of values which are both non-empty. A *state*  $s$  is a function from variables to values,  $s : V \rightarrow D$ . For state  $s$ , variable  $v$ , and value  $\alpha \in D$ ,  $s[v \leftarrow \alpha]$  denotes the state that is the same as  $s$  except  $s[v \leftarrow \alpha](v) = \alpha$ . Let  $S$  be the set of states. For a state  $s$  and a set of variables  $X$ ,  $s \upharpoonright X$  denotes the restriction of  $s$  to the variables in  $X$ .

A *transition*  $t$  consists of a *guard*  $g$  and an *action*  $a$ , where  $g$  is a predicate on states, and  $a$  is a function  $a : S \rightarrow S$ . We write  $g \rightarrow a$  to denote the transition with guard  $g$  and action  $a$ .

We can also form the action of a transition by composing two or more functions. If  $a, b : S \rightarrow S$ , then  $g \rightarrow a; b$  denotes the transition  $g \rightarrow \lambda s. b(a(s))$ .

We associate with each transition  $t = (g \rightarrow a)$  two sets of variables called the read and write sets of  $t$ ,  $R(t)$ ,  $W(t)$  respectively. The write set of  $t$  is the set of variables whose value can be changed by the transition in some state,  $W(g \rightarrow a) = \{v \mid \exists s \in S : g(s) \wedge a(s)(v) \neq s(v)\}$ . The read set of a transition is the set of variables that can affect either the enabling of the guard, or the value written to one of the variables in the write set,  $R(g \rightarrow a)$  is

$$\{v \mid \exists s \in S, \exists \alpha \in D : (g(s) \neq g(s[v \leftarrow \alpha])) \vee (a(s) \neq a(s[v \leftarrow \alpha]))\}$$

We say  $t$  *reads*  $v$  if  $v \in R(t)$ ;  $t$  *writes*  $v$  if  $v \in W(t)$ .

A *model* has the form  $(V, I, T, A)$ , where  $V$  is a set of variables,  $I$  is a set of initial states over  $V$ ,  $T$  is a set of

transitions, and  $A$  is a set of assertions (state formulas over  $V$ ) which can be empty.

### B. Executions

We consider two notions of execution for models. An *interleaving* execution of a model is based on steps in which a single enabled transition fires. For a transition  $t = (g \rightarrow a)$ , we write  $s \xrightarrow{t} s'$  to denote that  $g(s)$  holds and  $s' = a(s)$ . An interleaving execution of  $(V, I, T, A)$  is a sequence of states  $s_0, s_1, \dots$ , such that  $s_0 \in I$  and for all  $i \geq 0$ , there is a transition  $t_i \in T$  such that  $s_i \xrightarrow{t_i} s_{i+1}$ .

We now introduce the *concurrent* executions of a model. First we define the concurrent firing of a set of transitions. If  $t_1 = (g_1 \rightarrow a_1), t_2 = (g_2 \rightarrow a_2)$  are two transitions with  $W(t_1) \cap W(t_2) = \emptyset$ , we write  $s \xrightarrow{\{t_1, t_2\}} s'$  to denote that  $g_1(s)$  and  $g_2(s)$  hold, and  $s' = (a_1(s) \mid W(t_1)) \cup (a_2(s) \mid W(t_2)) \cup (s \mid V - W(t_1) - W(t_2))$ . In the preceding, we form the function  $s'$  by taking the union of restricted functions. We define the notion of the concurrent firing of a set  $E$  consisting of more than two transitions similarly, and write it as  $s \xrightarrow{E} s'$ .

The idea of a concurrent execution of a model is that all of the enabled transitions fire at each step. However, we have to deal with the problem of write conflicts between different transitions. This is because if two transitions  $t_i, t_j$  are both enabled at  $s$ , and they both write a variable  $v$ , then the value of  $v$  cannot be defined at the next state. For a state  $s$  such that two transitions  $t_1, t_2$  are enabled, with  $W(t_1) \cap W(t_2) \neq \emptyset$ , we say there is a write-write conflict at  $s$ . When a write-write conflict occurs at  $s$ , for the ease of exposition, we will define execution to simply stay at  $s$  (we can also define this as an error). Thus, a concurrent execution of a model  $(V, I, T, A)$  is a sequence of states  $s_0, s_1, \dots$ , such that  $s_0 \in I$  and for all  $i \geq 0$ ,  $s_i \xrightarrow{E_i} s_{i+1}$ . Here,  $E_i$  is the set of *all* transitions (maximal set) enabled at state  $s_i$  if there is no write-write conflict at  $s_i$ , i.e.  $s_i \xrightarrow{E_i} s_{i+1}$ . Otherwise,  $s_i = s_{i+1}$ .

Now we define *labelled* executions. Given an execution  $S_e = s_0, s_1, \dots$  of a model, a labelled execution of  $S_e$  is a sequence:  $s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \dots$ . If  $S_e$  is an interleaving execution, then for each  $i \geq 0$ ,  $E_i$  is a transition with  $s_i \xrightarrow{E_i} s_{i+1}$ . Otherwise,  $E_i$  is a set of transitions (defined, as above, to be maximal set) such that  $s_i \xrightarrow{E_i} s_{i+1}$ , or  $s_i = s_{i+1}$ . In executions we may write  $t$  for the singleton set of transitions  $\{t\}$  if the context makes the intended usage clear.

### C. Annotations

A transition may be annotated by associating formulas for pre- and post-conditions with the transition. We write  $g \rightarrow \{P\}a\{Q\}$  to denote the annotation of a transition  $g \rightarrow a$  with precondition  $P$  and postcondition  $Q$ . The formulas  $P, Q$  are formulas over the variables of the model. We can omit  $\{P\}$  or  $\{Q\}$  in the cases when it is equivalent to  $\{true\}$ . For an execution  $S = s_0 \xrightarrow{E_0} s_1 \xrightarrow{E_1} \dots$ , we say that  $S$  satisfies the annotation  $g \rightarrow \{P\}a\{Q\}$ , if whenever  $s_i \xrightarrow{E_i} s_{i+1}$  and  $E_i$  contains (if  $S$  is a concurrent execution) or is (if  $S$  is an interleaving execution) the transition  $g \rightarrow a$ , then  $P(s_i)$  and

$Q(s_{i+1})$  both hold. The intuition is that if a transition fires in an execution, then both the precondition and the postcondition must hold. An annotated model is one which has one or more annotated transitions.

Let  $M = (V, I, T, A)$  be a model and  $p$  be a formula. For an execution (either interleaving or concurrent)  $S_e = s_0, s_1, \dots$  of  $M$ , we say  $S_e$  satisfies  $p$  if  $p(s_i)$  holds for every  $i \geq 0$ . If  $M$  is a model in which only interleaving executions are considered, we say  $M$  satisfies  $p$  if every interleaving execution of the model satisfies  $p$ . We denote this by  $M \models p$ . Similarly, we define  $M \models p$  for a model  $M$  where only concurrent executions are considered. For an annotated transition  $t$  of  $M$ , we also write  $M \models t$  to denote that every execution of  $M$  satisfies  $t$ . Finally, if  $M$  satisfies all the assertions of  $A$  and its set of annotations then  $\models M$  holds.

### D. Transactions

We are interested in showing the correspondence between implementation and specification models, where the implementation may take many steps to accomplish the work of one step in the specification. To help define the correspondence between an implementation and specification, we introduce the notion of a *transaction*.

The purpose of a transaction is to collect the implementation transitions that correspond to a single transition of the specification. Each transaction has a unique state variable called the *alive* variable. In an initial state of the model, all of the alive variables for transactions are set to false. At this state, no transitions in a transaction are enabled until a transition called the *trigger* of the transaction is fired. If the transaction takes multiple steps to finish, then the alive variable is set to true. Once the guard of the trigger transition is enabled, other transitions in the transaction can fire. The other transitions can continue to fire until a *closing* transition resets the alive variable to false. Such a closing transition is assumed to always exist, and it could be the same as the trigger.

We now describe transactions more precisely. A transaction is a set of transitions, formed as follows. Let  $t = (g \rightarrow a)$  be a transition, and  $T$  be a possibly empty set of transitions. Then  $\text{transaction}[t; T]$  is the following set of transitions:

$$\{(g \wedge \neg \text{alive}) \rightarrow (a; \text{alive} \leftarrow \text{MultiCycle})\} \cup \cup_{(g_i \rightarrow a_i) \in T} \{(g_i \wedge (g \vee \text{alive})) \rightarrow a_i\}$$

In the above formula, *alive* is the alive variable unique to this transaction, and  $t$  is the trigger transition. The first half of the formula says that the trigger transition fires when its guard  $g$  is true and the alive variable is false. Firing the trigger transition updates the state by the action  $a$ . The function *MultiCycle* is true in a state if the transaction will take more than one cycle. The alive variable is set to true if the transaction takes multiple steps.

A transition in  $T$  fires when its guard  $g_i$  is true and either (i) the alive variable is false and the trigger guard is true, or (ii) the alive variable is true. That is, transitions in  $T$  are permitted to fire on the same step as the trigger transition.

As said earlier, transactions provide a way of organizing the transitions that realize the operations by the hardware

into mutually exclusive sets of transitions. However, note that during execution, the execution semantics are defined in terms of the concurrent executions generated by  $\cup_i U_i$  where  $U_i$  is a transaction. This means that transitions from different transactions can fire concurrently.

In the sequel, we study read-write relations between concurrent transactions in a way that is analogous to concurrency and variable access in threaded programs. We define the read and write sets of a transaction  $U_i$  to be  $R(U_i) = \cup_{t \in U_i} R(t)$ ,  $W(U_i) = \cup_{t \in U_i} W(t)$ , respectively.

### E. Implementation Model

We define an *implementation* model, or low-level model, to be a model  $M_L = (V_L, I_L, T_L, A_L)$ , where  $T_L$  is the union of a set of transactions,  $T_L = \cup_i U_i$ , and the transactions  $U_i$  are pairwise disjoint sets of transitions. This is the structure that we will assume for implementations.

We say that a variable  $v$  is *active* in state  $s$  in an implementation model, if there is a transaction  $U$  with  $v \in W(U)$ , and the alive variable for  $U$  is true in  $s$ . For a state  $s$ , let  $\text{active}(s)$  be the set of all variables active in  $s$ , and  $\text{inactive}(s)$  be the set of model variables not in  $\text{active}(s)$ .

## III. CORRECT IMPLEMENTATION

We consider three kinds of variables in the implementation model and discuss how each kind of variable corresponds to the specification. An *interface variable* must match the specification model at all times. A *transactional variable* must match the specification model except when the variable is active. The specification for a transactional variable is comparable to a resource invariant in a shared variable program with critical sections [21]. The similarity is that a transactional variable in our theory and a shared variable used in critical sections [21] both have a specified value when the variable is not being updated. Finally, the implementation can have additional variables that are not present in the specification and are not constrained.

Let  $M_H = (V_H, I_H, T_H, A_H)$  be a model and  $M_L = (V_L, I_L, T_L, A_L)$  be an implementation model over variables  $V_H, V_L$  respectively, where  $V_H \subseteq V_L$ . Let  $V_I \subseteq V_H$  be the set of interface variables and  $V_T \subseteq V_H$  be the set of transactional variables. We say that  $M_L$  implements  $M_H$  with interface variables  $V_I$  and transactional variables  $V_T$  if for every concurrent execution  $l_0, l_1, \dots$ , of  $M_L$ , there exists an interleaving execution  $h_0, h_1, \dots$ , of  $M_H$ , and an increasing sequence of natural numbers  $n_0 < n_1 < \dots$  such that for all  $i \geq 0$ ,

$$\begin{aligned} & (l_i \mid V_I = h_{n_i} \mid V_I) \\ \wedge & (l_i \mid (V_T \cap \text{inactive}(l_i)) = h_{n_i} \mid (V_T \cap \text{inactive}(h_{n_i}))). \end{aligned}$$

In this definition, step  $i$  of the implementation is matched in the specification at step  $n_i$ , and variables in  $V_I$  plus those in  $V_H$  that are inactive must have the same value in both models. If  $M_L$  implements  $M_H$ , then whenever  $M_L$  reaches a cleanly halted state with no transactions in progress, all of the variables in  $V_I$  and  $V_T$  must be in a state reachable in the specification  $M_H$ .

This definition can be augmented with additional invariant assertions to express stronger properties. For example, one invariant could be that  $n_i$  is the number of transactions in  $M_L$  which have completed at  $l_i$ . Another example could be that for all the transitions which are fired between  $h_{n_i}$  and  $h_{n_{i+1}}$ , their write sets are disjoint.

**Proposition 1.** Suppose  $M_H$  is a model and  $M_L$  is an implementation model such that  $M_L$  implements  $M_H$  with  $V_I$  and  $V_T$ . If  $M_H \models P$ , for an invariance property  $P$ , then  $M_L \models \text{inactive}_P \Rightarrow P$ , where  $\text{inactive}_P$  is a formula that asserts that all the transactional variables in  $\text{vars}(P)$  are inactive. The formula  $\text{inactive}_P$  can be expressed using the alive variables of transactions.

## IV. MONOLITHIC MODEL CHECKING

The refinement of the specification model  $M_H$  by the implementation model  $M_L$  can be checked in a straightforward way by constructing a *monolithic checking* model, called  $M_{MC}$ .  $M_{MC}$  defines all the possible executions of the implementation, and for each execution, asserts that the states reached by the implementation can also be reached through specification transitions. In effect,  $M_{MC}$  executes a cross product construction. In Section V, we describe a compositional checking method that allows much of the checking to be performed on smaller models.

Let  $M_H = (V_H, I_H, T_H, A_H)$  be a specification model and  $M_L = (V_L, I_L, T_L, A_L)$  be an implementation model, where  $V_H \subseteq V_L$ . Let  $V'_H$  be a fresh set of variables, with one new variable corresponding to each variable in  $V_H$ . For a set of variables  $V$ ,  $V'$  is the result of replacing all variables from  $V_H$  in  $V$  with the corresponding primed variable. That is, for  $V \subseteq V_L$ ,  $V' = \{v' \mid v \in V \cap V_H\} \cup (V - V_H)$ . For  $V \subseteq V_L - V_H$ ,  $V' = V$ . In addition, priming of expressions is defined to be the priming of the free variables of the expression.

We define the monolithic model as follows,  $M_{MC} = (V_L \cup V'_H, I_{MC}, T_{MC}, A_{MC})$ . The variables  $V_L$  will be updated according to the implementation model  $M_L$ , while the variables  $V'_H$  will be updated according to the specification model  $M_H$ .

We will check that for each initial state of the implementation, there exists an initial state of the specification that matches on the variables in  $V_H$ , that is,  $(I_L \mid V_H) \subseteq I_H$ . Then we define the initial states of  $M_{MC}$  to be  $I_{MC} = \{s_L \cup (s_L \mid V_H)'\} : s_L \in I_L$ . Thus, for each initial state  $s_L$  of  $M_L$ , we start  $M_{MC}$  in the state consisting of  $s_L$  and a primed copy of the state  $s_L$  over the variables in  $V'_H$ .

Our method is based on showing that for each transaction  $U_i$ , there is corresponding transition  $u_i$  in the specification such that  $U_i$  implements  $u_i$ . We call  $u_i$  the specification transition of  $U_i$ , written  $\text{spec}(U_i)$ .

We assume the specification always contains a stuttering transition of the form  $\text{true} \rightarrow id$ , where for all states  $s$ ,  $id(s) = s$ . For a transaction  $U_i$  that only writes to variables in  $V_L - V_H$ , we can assign the specification transition to be the stuttering transition.

We check refinement by finding for each transaction  $U_i$  a transition  $c_i$  in  $U_i$  that acts as a *commit* transition in the sense that whenever  $c_i$  fires, the specification transition  $u_i$  is

enabled on the primed variables. This allows the model  $M_{MC}$  to fire the specification transition  $u_i$  on the primed variables whenever  $c_i$  fires.

The transitions of  $M_{MC}$  are defined as follows. For each transaction  $U_i$ , let the commit transition  $c_i$  be  $cguard_i \rightarrow cact_i$ . To build the cross product model, we replace  $c_i$  with a transition that executes  $c_i$  on the unprimed variables while executing  $spec(U_i)$  on the primed variables. Let  $spec(U_i)$  be  $sguard_i \rightarrow sact_i$ . The transition

$$cguard_i \rightarrow \{sguard'_i\}sact'_i; cact_i$$

has the same guard as the commit transition, and uses a precondition annotation to check that the guard of the specification transaction is enabled on the primed variables. When this transition fires, it executes the action  $sact_i$  on the primed variables and executes  $cact_i$  on the unprimed variables. For each transaction  $U_i$  of  $M_{MC}$ , we call the precondition  $sguard_i$  in the above formula the specification *enableness* condition of  $U_i$ .

Using the above transition, we can define the transitions of  $M_{MC}$  to be

$$\bigcup_i (U_i - \{c_i\} \cup \{cguard_i \rightarrow \{sguard'_i\}sact'_i; cact_i\}).$$

To check the implementation relation, we define assertions in  $M_{MC}$ . We check the interface variables by defining the invariant assertions of  $M_{MC}$  to be  $A_{MC} = \{v = v' \mid v \in V_I\}$ . These assertions check that the implementation always matches the specification on interface variables.

For transactional variables, we check the implementation relation as follows. Consider a transaction  $U_i$  and a transactional variable  $v$ . If  $v$  is written by either  $U_i$  or the specification transition  $spec(U_i)$ , then we assert that  $v = v'$  holds at the end of the transaction. We implement these checks by adding a postcondition annotation of the form

$$\{\neg alive_i \Rightarrow vars_i = vars'_i\},$$

to each transition of transaction  $U_i$ . In this formula,  $alive_i$  is the alive variable of  $U_i$  and  $vars_i$  is defined to be the vector of all variables written by either a transition of  $U_i$  or written by  $spec(U_i)$ . Attaching this postcondition to each transition  $t$  of  $U_i$  has the following meaning: if the transaction  $U_i$  completes on a given step of the execution of  $M_{MC}$ , then  $alive_i$  will be false in the following state. The postcondition asserts that  $vars_i$  must equal  $vars'_i$  in this state. These assertions check the necessary conditions for the transactional variables.

*Checking Correct Implementation:* Let  $S_{mc} = mc_0, mc_1, \dots$ , be an execution of  $M_{MC}$ . If all the specification enableness conditions are satisfied in this execution, one or more specification transitions can fire at each  $mc_i$ . It follows that  $mc_0 \mid V'_H, mc_1 \mid V'_H, \dots$ , is a *concurrent* execution of the specification. For  $i \geq 0$ , let  $h_i = mc_i \mid V'_H$ . Also, let  $S_h$  be the labelled concurrent execution  $h_0 \xrightarrow{E_0} h_1 \xrightarrow{E_1} \dots$ , where for  $i \geq 0$ ,  $E_i = \{u_{i,1}, \dots, u_{i,n_i}\}$  is the set of specification transitions that fire simultaneously at state  $h_i$  in the projected execution  $S_h$ .

We would like to know when the concurrent execution of the specification transitions in  $S_h$  can be converted into an interleaving execution. First, there cannot be a write-write conflict in  $E_i$ , for any  $i$ . Second, for each  $i$ , the concurrent firing of the transitions in  $E_i$  must be equivalent to some sequential firing order. For any two transitions  $t_1, t_2$ , say  $t_1$  must precede  $t_2$ , written  $t_1 \prec t_2$ , if  $R(t_1) \cap W(t_2) \neq \emptyset$ . For each set  $E_i$ , define  $\prec_i^*$  to be the transitive closure of the  $\prec$  relation on the transitions in  $E_i$ . If the relation  $\prec_i^*$  is irreflexive, then there is a sequential firing of the transitions in  $E_i$  that updates the state from  $h_i$  to  $h_{i+1}$ . If the relation  $\prec_i^*$  is irreflexive for all  $i$ , we say *serializability* holds for  $M_{MC}$ .

The following theorem states the necessary conditions for checking the implementation relation.

**Theorem 1.** Suppose  $M_{MC}$  is the monolithic model for implementation  $M_L$  and specification  $M_H$ . If (i)  $(I_L \mid V_H) \subseteq I_H$ , (ii) no write-write conflicts exist at any state of  $M_{MC}$ , (iii) serializability holds for  $M_{MC}$ , (iv)  $W(spec(U_i)) \subseteq W(U_i)$  holds for each transaction  $U_i$  in  $M_L$ , and (v)  $\models M_{MC}$ , then  $M_L$  implements  $M_H$  with interface variables  $V_I$  and transactional variables  $V_T$ .

## V. COMPOSITIONAL MODEL CHECKING

This section presents results that allow one to reason efficiently about a model by checking properties of a set of smaller, more abstract models. There are two basic ideas: abstraction and assume-guarantee reasoning.

Consider the problem of making an abstract model that conserves correctness of the annotations on a transition  $t$  in a model  $M$ . We would like to remove as many transitions from the model as possible, while maintaining the property that if the annotations hold in the abstract model, they hold in the full model.

To form an abstraction, we analyze the sources that supply values for variables that are read by a transition. To make a model more abstract, we can convert some sources into free input variables and remove certain transitions from the model. We will develop a condition that ensures that input sources for a transition in an abstract model are at least as general as the sources in the full model. One distinctive feature of our approach is that we allow abstractions in which a variable is abstracted to an input variable in some transitions, but the original variable is retained in other transitions. This approach allows us to control the level of detail contained in an abstract model. We use this control of abstraction to reason compositionally about transactions.

To form abstract models, we introduce *input variables*, and models with input variables. A model with input variables has the form  $M = (V, \tilde{V}, I, T, A)$ , where  $V$  is the set of state variables and  $\tilde{V}$  is the set of input variables. For each variable  $v$  in  $V$ , there are two corresponding input variables,  $\tilde{v}$  and  $\tilde{v}^n$ . The variable  $\tilde{v}$  is used to provide an unconstrained input on the current state. The variable  $\tilde{v}^n$  is used in annotated transitions, to provide an unconstrained input for postcondition annotations, which are evaluated in the “next” state of the model.

Semantically, a model with input variables is similar to a model without inputs. The states of the model map  $V \cup \tilde{V}$  to

values. Transitions can read but not write the input variables. The action function of a transition defines the next state of the ordinary state variables in  $V$  only. The next-state relation for input variables is unconstrained: an input variable can take on any value in the next state of a computation.

Without loss of generality, we can assume that all models have input variables, since a model that does not mention input variables can be represented as a model with input variables.

#### A. Abstraction

Let  $t = (g \rightarrow a)$  be a transition of a model  $M$ . We define a *read variant* of  $t$  as the result of replacing reads of zero or more variables in  $R(t)$  with their corresponding input variables. (Formal definition in Appendix.) We write  $\tilde{t}$  for read variant of  $t$ . A transition  $\tilde{t}$  is a read variant of itself in the case that no variables are renamed. If  $T$  is a set of transitions, then we define *variants*( $T$ ) to be the set of all read variants of transitions in  $T$ .

We now discuss our approach to abstraction in more detail. Let  $M = (V, \tilde{V}, I, T, A)$  be a model. We are interested in models built from a subset of *variants*( $T$ ) that contains no more than one read variant of any transition in  $T$ . Let us call such a set *consistent*. Intuitively, a model containing a consistent set of read variants of  $T$  can execute the behavior of  $M$  on a subset of the variables, without introducing two transitions that generate write-write conflicts at all states.

We want to define a syntactic condition that says when a set of read variants is capable of executing all behaviors of the full model over a subset of the state variables. Let  $\tilde{T}$  be a set of read variants of transitions in  $T$ . Consider  $t \in T$ ,  $\tilde{t}$  which is a read variant of  $t$ , and a variable  $v$  that is read by  $\tilde{t}$ . We say that the read of  $v$  in  $\tilde{t}$  is *sufficiently general* with respect to  $M$  if one of the following holds:

- 1)  $v$  is an input variable, or
- 2)  $\tilde{T}$  contains a read variant of every transition in  $T$  that writes to  $v$ , or
- 3) There is a transition  $s \in T$  that writes  $v$ ,  $\tilde{s} \in \tilde{T}$ , and in every concurrent execution of  $M$ , whenever  $t$  fires,  $s$  fires on the previous step.

We say that  $\tilde{T}$  is *sufficiently general* with respect to  $M$  if every read of every variable in  $\tilde{T}$  is sufficiently general with respect to  $M$ .

Our abstract models overapproximate an original model by reading the values of a combination of state and input variables. In order to compare executions of abstract models to the original models, we define the *effective input state* of a model with input variables at a given state. When a transition reads the value of an input variable  $\tilde{v}$ , the effective input state assigns the same value to the original variable  $v$ . Let  $M = (V, \tilde{V}, I, T, A)$  be a model and  $s$  be a state of  $M$ . The effective input state of a transition  $t$  at a state  $s$ ,  $eff(t, s)$ , is defined for  $v \in R(t)$ :

- 1) If  $t$  reads the state variable  $v$ , then  $eff(t, s)(v) = s(v)$ .
- 2) If  $t$  reads the input variable  $\tilde{v}$ , then  $eff(t, s)(v) = s(\tilde{v})$ .

The effective read states of a transition  $t$  in a model  $M$  are defined by  $Eff(t, M) =$

$\{eff(t, s) \mid s \text{ is a reachable state of } M \text{ and } t \text{ is enabled at } s\}$

The following proposition describes how a model  $M'$  containing a sufficiently general subset of read variants of transitions of a model  $M$  can overapproximate the behavior of  $M$ .

**Proposition 2.** Let  $M = (V, \tilde{V}, I, T, A)$  and let  $T'$  be a subset of  $T$ . Let  $M' = (V, \tilde{V}', I', \tilde{T}', A')$  be a model where  $I \subseteq I'$ , and  $\tilde{T}'$  contains a read variant of each transition in  $T'$ . Suppose  $\tilde{T}'$  is sufficiently general with respect to  $M$ . Then for all transitions  $t \in T'$   $Eff(t, M) \subseteq Eff(\tilde{t}', M')$ .

The notions of read variant and sufficiently general abstraction extend readily to models with annotations. A read variant of an annotated transition  $g \rightarrow \{P\}a\{Q\}$  is formed by consistently replacing read variables  $v$  with the corresponding same-state input variables  $\tilde{v}$  in  $g, P$  and  $a$ , and replacing variables  $v$  in  $Q$  with the corresponding next-state variables  $\tilde{v}^n$ .

Consider an annotated model  $M$  and a set  $\tilde{T}$  of read variants of transitions in  $M$ . A variable instance  $v$  in the precondition of an annotated transition is sufficiently general with respect to  $M$  under the same conditions that apply for a read instance in a guard or action. A variable instance  $v$  in a postcondition of a transition  $\tilde{t}$  is sufficiently general if one of the following holds:

- 1)  $v$  is a next-state input variable.
- 2)  $\tilde{T}$  contains a read variant of every transition of  $M$  that writes to  $v$ .
- 3)  $\tilde{t}$  writes to  $v$ .
- 4) There is a transition  $s$  of  $M$  that writes to  $v$ ,  $\tilde{s} \in \tilde{T}$ , and in every concurrent execution of  $M$ , if  $t$  fires then  $s$  fires on the same step.

As before, we say that  $\tilde{T}$  is *sufficiently general* with respect to  $M$  if every read of every variable in  $\tilde{T}$  is sufficiently general with respect to  $M$ .

**Proposition 3.** Let  $M = (V, \tilde{V}, I, T, A)$  and  $M' = (V, \tilde{V}', I', \tilde{T}', A')$  be models where  $I \subseteq I'$ , and  $\tilde{T}'$  is a subset of *variants*( $T$ ) that is sufficiently general with respect to  $M$ . Let  $t$  be an annotated transition in  $T$  and  $\tilde{t}'$  be a read variant in  $\tilde{T}'$ . Then  $M' \models \tilde{t}'$  implies  $M \models t$ .

#### B. Assume-Guarantee Reasoning

In the previous section, we developed a set of sufficient conditions under which the annotations of a transition can be checked in an abstract model. Assume-guarantee reasoning involving the input variables of an abstract model can reduce the size of the models that need to be checked below the size that is needed when abstraction is used alone.

If  $P(\tilde{V})$  is a satisfiable formula over the input variables in  $\tilde{V}$  and  $Q(V)$  is an invariance property over the state variables  $V$ , then define  $M; P(\tilde{V}) \models Q(V)$  to be true if  $Q(V)$  holds on all reachable states of the model  $M$ , provided the input variables are constrained at each step to satisfy  $P(\tilde{V})$ .

In its simplest form, assume-guarantee reasoning is a form of induction involving the input variables. Suppose  $M = (V, \emptyset, I, T, A)$  is a model, and  $M' = (V, \tilde{V}', I', \tilde{T}', A')$  is a model with input variables, where  $\tilde{T}'$  contains a read variant of every transition in  $T$ , and  $I \subseteq I'$ . Then the following scheme is valid. Note that any update of the variables  $V$  from the input

variables  $\tilde{V}$  requires at least one time step for a transition to fire.

$$\frac{M'; P(\tilde{V}) \models P(V)}{M \models P(V)}$$

The actual form of assume-guarantee reasoning that we use can be stated as follows. Suppose  $M = (V, \emptyset, I, T, \emptyset)$  is a model, and  $M_1 = (V, \tilde{V}, I_1, T_1, \emptyset)$ , and  $M_2 = (V, \tilde{V}, I_2, T_2, \emptyset)$  are models with input variables where  $T_1$  and  $T_2$  are subsets of  $\text{variants}(T)$  that are both sufficiently general with respect to  $M$ . We require that for each transition  $t$  in  $M$ , either  $M_1$  or  $M_2$  contains a read variant of  $t$ . Also suppose the initial states of  $M_1, M_2$  both contain the initial states of  $M$ . These conditions assure that  $M_1$  and  $M_2$  are abstractions of  $M$  that preserve the correctness of annotations on transitions in  $M$ .

We can use compositional assume-guarantee reasoning as follows. Let  $P(X)$  be a formula over a set of state variables  $X \subseteq V$ . Suppose that every transition in  $M$  that writes to a variable in  $X$  is annotated with the postcondition  $P(X)$ . Then we can use the following scheme, which checks that  $P(X)$  or a read variant is true in the initial state and whenever one of the variables in  $X$  is written. The scheme assumes  $P(\tilde{X})$  is always true over the input variables, where  $\tilde{X}$  is the result of replacing each variable in  $X$  with the corresponding input variable.

$$\frac{\begin{array}{l} I_1 \models P(X) \quad M_1; P(\tilde{X}) \models T_1 \\ I_2 \models P(X) \quad M_2; P(\tilde{X}) \models T_2 \end{array}}{M \models T}$$

As noted in the next section, in some cases the actual annotations to be checked can be simplified. For instance it may be the case that  $M$  consists of two transactions that can be shown to never overlap, and  $M_1$  and  $M_2$ , respectively, contain read variants of these two transactions. In this case, only the final value produced by each transaction can be read by the other transaction. It is not necessary to annotate the non-final transitions in this case.

Another case in which the invariant can be simplified is when no transition in one of the submodels,  $M_1$  or  $M_2$ , writes to a certain variable. In the next section, detailed implementation rules are described and the rules are specialized to consider the case of checking models having the form of the monolithic checking model  $M_{MC}$ .

## VI. IMPLEMENTATION

Now we describe an implementation of the compositional approach for the monolithic model  $M_{MC}$ .

For each transaction  $U_i$  in  $M_{MC}$ , we build an abstract model  $M_i = (V_{MC}, \tilde{V}_{MC}, I_i, T_i, A_i)$  that contains the correctness conditions for  $U_i$ . In this model,  $T_i$  is a set of read variants of transitions of  $U_i$ . For the initial states,  $I_i$  is highly unconstrained compared with  $I_{MC}$ .  $I_i$  covers all the states of  $M_{MC}$  at which the trigger transition of  $U_i$  fires in  $M_{MC}$ . Essentially,  $I_i$  consists of those states where the variables in  $V_H$  that  $U_i$  reads or writes, match with their correspondents in  $V'_H$ .

Now we discuss the variables whose reads are replaced with input variables in  $M_i$ . This is based on how the transaction  $U_i$  can interact with other transactions in  $M_{MC}$ .

- First, we consider variables  $v$  in  $V_L - V_H$ . If  $U_i$  reads  $v$ , another transaction writes  $v$ , and these transactions can overlap in  $M_{MC}$ , i.e. one starts before the other is completed, then we replace each read of  $v$  in  $M_i$  with the input variable  $\tilde{v}$ .
- Second, for variables  $v$  in  $V_H$ , if  $U_i$  reads  $v$ , another transaction writes  $v$ , and they can overlap in  $M_{MC}$ , then we replace the reads of  $v$  with  $\tilde{v}$  in  $M_i$ . At the same time, if  $v'$  is in the read set of the specification transition of  $U_i$ , then we also replace the reads of  $v'$  with  $\tilde{v}'$ , and we assume  $\tilde{v} = \tilde{v}'$  holds at every step of  $M_i$ . The reason to make these assumptions is that, to verify the specification enablement condition and the postcondition  $\text{vars}_i = \text{vars}'_i$  of  $U_i$  in  $M_i$ , reads of individual input variables would otherwise be too unconstrained. We will discuss how these assumptions will be guaranteed in the next.
- Third, for variables  $v$  in  $V_H$ , if  $U_i$  writes  $v$ , another transaction reads  $v'$  or writes  $v$ , and they can overlap in  $M_{MC}$ , then we make  $v = v'$  a postcondition of every transition that writes to  $v$  or  $v'$  in  $M_i$ . That is, we request that in every state of  $M_i$ ,  $v = v'$  holds. This is a guarantee part of assume-guarantee reasoning. In fact, the postcondition  $\text{vars}_i = \text{vars}'_i$  of  $U_i$  in  $M_i$  is another part of guarantee.

The above process describes one implementation of how we can decompose the model checking of  $M_{MC}$  into the checks of the  $M_i$ . It is straightforward that this implementation is just a special case of the abstraction and assume-guarantee reasoning which are presented in Section V. So all the abstract models built this way have the following property: if for all  $i$ ,  $M_i; P_i(\tilde{V}, \tilde{V}') \models T_i$  holds, then  $\models M_{MC}$  holds. Here,  $P_i(\tilde{V}, \tilde{V}')$  is the set of assumptions which are described in the second case.

In the above, if the check of some  $M_i$  fails due to abstraction using input variables, we can undo the replacements. In more detail, we can change every read of an input variable  $\tilde{v}$  in  $M_i$  back to reads  $v$ . We can then transitively include all the transactions of  $M_{MC}$  that write  $v$ , to  $M_i$ . All these are just different heuristics of implementing the compositional approach.

In order to use the compositional approach, we still need to deal with write-write conflicts, the serializability condition, and how to find out if two transactions can overlap in  $M_{MC}$ . Our current approach is to check these properties directly in  $M_{MC}$ . We first statically find out all the potential write-write conflicts in  $M_{MC}$ . We then check whether each pair of transactions with a statically detected potential write-write conflict can overlap in  $M_{MC}$ . If there is no overlap, the potential conflicts cannot occur. Otherwise, we check the occurrence of conflicts individually in  $M_{MC}$ . The serializability condition can be checked similarly. Finally, for all the transactions which don't involve statically detected write-write conflicts, we assume they can overlap with each other (clearly this is sound).

## VII. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Given a specification model  $M_H$  in Murphi and an implementation model  $M_L$  in HMurphi, we first build a monolithic model  $M_{MC}$ . We then use a static analyzer to find out all the possible shared variables read-write and write-write usages in

$M_{MC}$ . Following that, we use *muv* to translate the model into VHDL. It assumes that each rule once enabled finishes in one clock cycle. Finally, we use a VHDL verifier, in our case the IBM RuleBase [22], to check the refinement properties.

### A. Benchmark Details

We use the German protocol [9] as the specification of our benchmark. This protocol is a simple invalidation-based directory protocol. A cache line can be in one of the states: *invalid*, *shared* and *exclusive*. Right now, the system models one address with one home node and one remote node.<sup>1</sup>

In the model each node contains a memory, a cache and a directory. To prevent communication deadlocks and allow non-FIFOed delivery of messages, multiple communication channels are introduced for each pair of nodes. As a whole, the specification model has about 500 LOC in Murphi.

The implementation model [8] is much more complex. Each node contains three controlling units: the local, home and remote units. It also contains a directory, a cache and a memory. Different units in one node can function simultaneously. Figure 2 shows the structure of a node in the implementation.

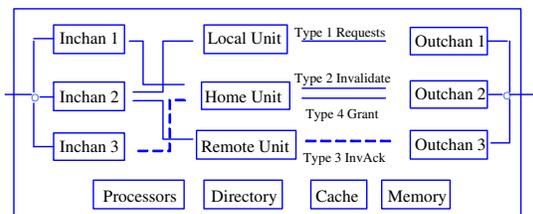


Fig. 2. Structure of an implementation node

In the implementation model, other than the two nodes, there also exists one router and six buffer units. The router is used to transfer messages from source nodes to destination buffers, and it can select at most one message to transfer at each cycle. The buffers are used to store messages in the destination side, before the destination nodes pick the messages and process them. Altogether, the monolithic model has about 1500 LOC in HMurphi.

### B. Transaction Details

In this benchmark, there are altogether 13 transactions including transactionsets. A *transactionset* is a set of transactions which only differ in certain parameters. The sequence of how the transitions within each transaction fire, is fixed in this example. These sequences fall into five categories, as shown in the following figure.

The first category represents the cases when a transaction has only one transition. The second represents the cases when a transaction has two transitions and they fire concurrently in one cycle. These two transitions are in different hardware modules, and they are structurally separate. They happen to

<sup>1</sup>These configurations are in part constrained by the VHDL verifiers we can access.

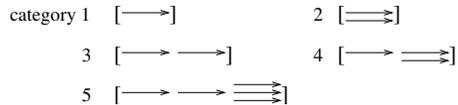


Fig. 3. Transition firing sequences of transactions.

fire concurrently in one cycle, so we cannot combine them into one transition. The rest is similar.

In this example, one transition can be in different transactions. In that case, each instance of the transition has a different *alive* variable and a trigger transition associated with it. It is worthwhile mention that all the transitions within a transaction do not necessarily need to fire in consecutive cycles. There could have gaps in between and all our proof rules still work. For example, in hardware because all the processing units can be active at the same time, a unit may have to wait several cycles to access a shared resource.

In our benchmark, different transactions can overlap. For example, one transaction in the fifth category is that a node which requested an exclusive copy, receives the copy. The five transitions in this transaction work as follows: (i) the local unit copies the replied message from the buffer unit, and sets the flag of availability for cache unit true; (ii) the cache unit copies the message, and sets the flag of acknowledgement for local unit true; (iii) the local unit resets the flag of availability, and marks the action of requesting a copy as completed; (iv) the cache unit resets the flag of acknowledgement; (v) the buffer unit resets the message. Another transaction in the first category is a node updating its cache line, when it has an exclusive copy. These two transactions can overlap in a way such that the second transaction fires immediately after the second transition of the first transaction. So when the first transaction is completed, the cache line data is from the second transaction, not from the replied message. The refinement still holds in this case, because when both transactions are finished, the resulting state can be reached in the specification by firing the two transactions sequentially. Our method shows that the refinement holds in this case.

### C. Experiment Results

For this benchmark, we have successfully verified that the implementation refines the specification, using both the monolithic and the compositional approaches. All the experiments were performed on a PC with an Intel Pentium CPU of 3.0GHz with 2GB memory. Table VII-C shows the experimental results of the two approaches where the datapath of the cache line is one bit.

	Compositional MC		Monolithic MC
	Conflict check	Abstract models	
# of FF	108	89	212
# of gates	5763	2194	8574
MC time	674	163	1036

In this table, “MC” represents model checking, and “# of FF” and “# of gates” represent the number of flip-flops and

gates of the boolean circuits, after the models under test were reduced with the same configurations. The model checking time is counted in seconds. The cost of the compositional approach comes from two parts. The first is checking the write-write conflicts and serializability on the monolithic model, and the second comes from model checking for each abstract model. Here, we only show the maximum boolean circuit on all the abstract models, and the total model checking time on them. There are altogether 13 abstract models, some of which are very small. The boolean circuits of the abstract models range from 18 to 89 flip-flops, 513 to 2194 gates, and the model checking time ranges from 12 to 52 seconds.

Moreover, for datapath larger than one bit, the monolithic approach exceeds the 300 flip-flops limit that our academic version of RuleBase can handle. In contrast, the compositional approach can handle upto 26-bit of datapath. From the above, we can see that the compositional approach can enable us to verify larger models, in which these models may exceed the capacity of the monolithic approach.

### VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we present a framework for modeling and verifying transaction based hardware protocol implementations. While our methodology is being developed with shared memory subsystems as driving examples, we believe that it is far more general in scope, and will very likely find other applications within a multicore design as well.

Our method is currently implemented as a manual form of abstraction and assume-guarantee reasoning. We plan to mechanize our methods as much as possible, to get the most benefit. It will be important to understand how to effectively combine our abstraction and assume-guarantee methods with automatic abstraction algorithms found in current model checkers.

As our future work, we would also like to explore how to reduce the cost of write-write conflicts and serializability checks of the compositional approach. We will also try our approaches on other applications such as pipelined systems, and with other verification tools.

### ACKNOWLEDGMENT

The anonymous referees made helpful comments.

### REFERENCES

- [1] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein, "System design methodology of UltraSPARC-I," in *Design Automation Conference*, 1995.
- [2] S. M. German, "Formal design of cache memory protocols in IBM," in *Formal Methods in System Design*, 2003.
- [3] C. T. Chou, S. M. German, and G. Gopalakrishnan, "Tutorial on specification and verification of shared memory protocols and consistency models," in *Formal Methods in Computer-Aided Design*, 2004.
- [4] B. Batson and L. Lamport, "High-level specifications: Lessons from industry," in *Formal Methods for Components and Objects*, 2002.
- [5] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *IEEE Intl. Conference on Computer Design: VLSI in Computers and Processors*, 1992.
- [6] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [7] S. M. German, personal Communication.
- [8] S. German and G. Janssen, "A tutorial example of a cache memory protocol and RTL implementation," IBM Research Report, RC23958, Tech. Rep., 2006.
- [9] S. German, "Tutorial on verification of distributed cache memory protocols," in *Formal Methods in Computer Aided Design*, 2004.
- [10] N. Mellergaard and J. Staunstrup, "Tutorial on design verification with synchronized transitions," in *International Conference on Theorem Provers in Circuit Design*, 1994.
- [11] J. Staunstrup, *A Formal Approach to Hardware Design*. Kluwer Academic Publishers, 1994.
- [12] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [13] Arvind and X. Shen, "Using term rewriting systems to design and verify processors," in *IEEE Micro*, 1999.
- [14] Arvind, R. Nikhil, D. Rosenband, and N. Dave, "High-level synthesis: An essential ingredient for designing complex ASICs," in *International Conference on Computer Aided Design*, 2004.
- [15] J. Hoe and Arvind, "Hardware synthesis from term rewriting systems," in *International Conference on Very Large Scale Integration: Systems on a Chip*, 1999.
- [16] [Http://www.bluespec.com/](http://www.bluespec.com/).
- [17] S. Park, "Computer assisted analysis of multiprocessor memory systems," Ph.D. dissertation, Stanford University, 1996.
- [18] C. Jones, "Tentative steps towards a development method for interfering programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 596–616, 1983.
- [19] K. McMillan, "A compositional rule for hardware design refinement," in *Computer Aided Verification*, 1997.
- [20] S. Hazelhurst and C. H. Seger, "Symbolic trajectory evaluation," in *Formal Hardware Verification – Methods and Systems in Comparison*, 1996.
- [21] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Communications of the ACM*, vol. 19, no. 5, pp. 279–284, 1994.
- [22] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal, "Rulebase: Model checking at IBM," in *Intl. Conference on Computer-Aided Verification*, 1997.

Appendix. Definition of read variant. Let  $V_t = \{v_1, \dots, v_n\} \subseteq V$ ,  $n \geq 0$ . We define a substitution function  $\sigma$ , to replace each read of  $v \in V_t$  in  $t$  with the input variable  $\tilde{v}$ :

$$\sigma(t, V_t) = (\sigma(g, V_t) \rightarrow \sigma(a, V_t))$$

where

$$\sigma(g, V_t) = \lambda s. g(s[v_1 \leftarrow s(\tilde{v}_1)] \dots [v_n \leftarrow s(\tilde{v}_n)])$$

$$\sigma(a, V_t) = \lambda s. a(s[v_1 \leftarrow s(\tilde{v}_1)] \dots [v_n \leftarrow s(\tilde{v}_n)])$$

Then  $\sigma(t, V_t)$  is a read variant of  $t$ .

# Automating Hazard Checking in Transaction-Level Microarchitecture Models

Yogesh Mahajan and Sharad Malik

Department of EE, Princeton University, Princeton, NJ 08544, USA

Email: {yogism, sharad}@princeton.edu

**Abstract**—Traditional hardware modeling using RTL presents a time-stationary view of the design state space which can be used to specify temporal properties for model checking. However, high-level information in terms of computation being performed on units of data (transactions) is not directly available. In contrast, transaction-level microarchitecture models view the computation as sequences of (data-stationary) transactions. This makes it easy to specify properties which involve both transaction sequencing and temporal ordering (e.g. data hazards). In RTL models, additional book-keeping state must be *manually* introduced in order to specify and check these properties. We show here that a transaction-level microarchitecture model can help automate checks for such properties via the automated creation of the book-keeping structures, and illustrate this for a simple pipeline using SMV. A key challenge in model-checking the transaction-level microarchitecture is representing the dynamic transaction state space. We describe an encoding as well as a fixed point computation for this.

## I. INTRODUCTION

Often, the functionality of a hardware design is specified at the level of computation on units of data (transactions). Consequently, one has to verify that the design satisfies properties which involve the correctness of individual transactions, as well as the correctness of interactions between transactions.

However, traditional models such as RTL take a time-stationary view of the design and do not provide an easy mechanism to associate events with the corresponding specification-level transactions. In contrast, transaction-level microarchitecture ( $\mu$ -architecture) models (e.g. [1]) make this information easily accessible for use in properties and formal analysis.

We argue that using a transaction-level  $\mu$ -architecture model for design (at a level higher than RTL) makes it easier to state interesting properties as well as to increase automation of some common verification tasks and thereby improve productivity. This paper focuses on the detection of data hazards in transaction-level  $\mu$ -architecture models to illustrate this point.

## II. TRANSACTION-LEVEL $\mu$ -ARCHITECTURE MODELING

A transaction-level  $\mu$ -architecture model ( $\mu$ -TLM) separates the computation on data from the usage of resources. Transactions describe computation on units of data present in a set of shared state elements. (The shared state includes the “architected state” of the specification level.)

A transaction type  $T$  is described as an FSM where the states of the FSM may be labeled with a set of resources. (The example in Fig 1(a) has resource R1 associated with state B.) All transaction instances of type  $T$  can read/write to the shared

state elements. In addition, each transaction instance of type  $T$  has an associated private copy of local state elements  $L^T$  (only accessible to that particular transaction instance). While the shared state elements are persistent across transactions, a fresh set of transaction local state elements is used by each transaction instance during its lifetime.

The transaction FSM is constrained as follows regarding when it can make transitions. A transition to a new state can be made only after the transaction instance gains possession of all the resources associated with the new state. The allocation of resources among transaction instances is done by *resource managers* which are modeled as FSMs. Thus, the resource managers drive the execution of the  $\mu$ -architecture (Fig 1(b)) while the transactions perform the actual computation on data.

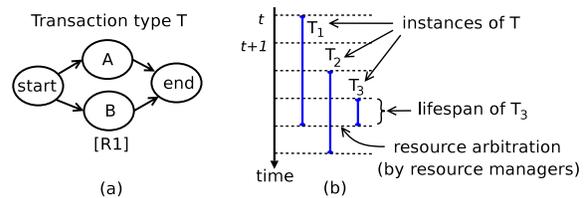


Fig. 1. Example of (a) transaction FSM (b) transaction scheduling

The resource manager FSMs maintain their own state throughout the execution of the system. The interaction between transaction instances and resource managers is that the transactions request resources which may be granted by the resource managers.

In this methodology, the RTL is synthesized from the  $\mu$ -TLM [1] followed by an equivalence check between the resulting RTL and the  $\mu$ -TLM.

## III. PROPERTIES INVOLVING TRANSACTION INFORMATION

Writing properties that relate data values across the duration of a transaction or properties that involve data interactions between multiple transactions is not easy to do for the RTL because this requires precise knowledge of how the transactions map to the RTL.

In a  $\mu$ -TLM, it is easy to group related events associated with individual transaction instances. Additionally, a  $\mu$ -TLM has an inherent notion of sequencing/ordering of transactions during their execution. These two characteristics make it simpler to state some interesting properties for the model.

As an example of an interesting property which involves both transaction ordering and temporal ordering, consider the property which checks that a design is free of data hazards: A read-after-write (RAW) hazard is present if there is a write to a memory location by transaction  $T_i$  which occurs later in time than a read from the same location by some transaction  $T_j$  with  $i < j$ . Here the *later* is a temporal ordering and  $i < j$  is a transaction ordering requirement of the property.

Note that there is no need to manually modify the model in order to be able to state the property, since the model has the notion of transactions and their ordering. *For an RTL model, the designer has to know how the specification-level transactions map to the RTL and manually add additional state or logic to make this information accessible. This gap in automation has been a limiting factor in property verification.*

Another property which requires read and write events to be associated with transaction level information involves relating the  $\mu$ -architecture level to the specification: A transaction reads the same inputs and produces the same outputs at the  $\mu$ -architecture level as at the specification-level. (This ‘correctness’ check is the subject of current work.)

#### IV. MODEL CHECKING ISSUES

Even though modeling the  $\mu$ -architecture as a  $\mu$ -TLM frees the designer of the burden of re-introducing transaction level information into a time-stationary model, it introduces some new challenges due to the dynamic nature of the state space of the model as explained below.

For the following discussion, we will consider a design with a single transaction type  $T$  for illustration, but similar comments also apply in the case of multiple transaction types.

##### A. State Space of the Model

The overall state of a  $\mu$ -TLM has two components:

- *Persistent state*: This consists of the shared state through which transactions interact and the state of the resource manager FSMs.
- *Transient state*: Each transaction instance maintains some state during its lifetime. The  $i^{\text{th}}$  instance of a transaction of type  $T$ , i.e.  $T_i$ , is associated with the state variables  $Q_i$  which consist of two components - the FSM encoding variables and the local state variables.

The transient state associated with transaction instances poses a problem for model checking. If each new transaction uses a fresh set of state variables, this will lead to an unbounded increase in the number of state variables. An encoding of the transient state which reuses the state variables of retired transactions for newer transactions instances might fail to retain transaction ordering information. Hence, a key challenge is the representation of the state space of the model in a form suitable for model checkers via a suitable encoding which preserves transaction ordering information.

##### B. Bounding the Number of Transient State Variables

If the most recent transaction instance of type  $T$  is  $T_n$ , then the transient state for type  $T$  transactions would consist of

the state variables  $Q_1 \cup Q_2 \cup \dots \cup Q_n$  assuming no reuse of state variables. However, the state of transactions which have already completed is not required in order to determine the next state of the system. The persistent state and the state of the currently active transactions are adequate for this. Suppose that  $\mathcal{A}^t$  is a set containing the instance numbers of only the active transaction instances at time  $t$  (those which have left the initial state but have not yet finished execution). Then, at time  $t$ , the transient state is represented by the variables  $\bigcup_{i \in \mathcal{A}^t} Q_i$ .

In practice, the number of concurrently active transactions  $|\mathcal{A}^t|$  will be bounded due to the limited availability of resources in hardware. In some cases, it may be possible to determine an upper bound on this number based on the transactions’ usage of resources, or by bounding the lifetime of transactions. Otherwise, we may assume an upper bound and then additionally verify that this assumption is correct.

With a bound  $|\mathcal{A}^t| \leq k$  on the number of active transactions, it becomes possible to fix the number of state variables needed to represent the transient state of the system at any possible instant. With a new set of state variables  $S_1, S_2, \dots, S_k$  and a mapping  $f^t : \mathcal{A}^t \rightarrow \{1, 2, \dots, k\}$  such that  $S_{f^t(i)} = Q_i$ , the transient state at time  $t$  is rewritten as  $S_1 \cup S_2 \cup \dots \cup S_k$ .

##### C. Preserving Transaction Ordering Information

We can choose the mapping  $f^t$  to make it easy to determine the relative ordering of when two transactions started just by looking at which  $S_i$  variables are used for their corresponding state variables, i.e. for  $i, j \in \mathcal{A}^t$  we can have  $(i < j) \Rightarrow (f^t(i) < f^t(j))$ . Such a mapping is maintained dynamically during the execution of the system.

Consider the simple case when at most one new active transaction instance may be created at any time, with  $|\mathcal{A}^t| \leq k$ . This is modeled by always having exactly one transaction instance attempting to leave the *start* state (become active), which is always mapped to the state variables  $S_{k+1}$ . The remaining active transaction instances use the state variables  $S_1, S_2, \dots, S_{|\mathcal{A}^t|}$  in the order that the transaction instances were instantiated. Variables  $S_{|\mathcal{A}^t|+1}$  through  $S_k$  are set to a sentinel value representing the *end* state. The transient state at time  $t$  is  $S_1 \cup S_2 \cup \dots \cup S_{k+1}$ . Each state update involves two steps. First, we compute the next state function of each of the  $|\mathcal{A}^t| + 1$  transaction FSMs, based on their interactions with the resource managers. Second, we have another combinational computation which remaps the next states onto the appropriate state variables  $S_i$  for the next time step based on the progress of the current transaction instances. This remapping happens when some transaction instance ends, thus freeing up its  $S$  variables, or when a new transaction instance becomes active. Note that such an arrangement implies that the same transaction instance may actually use different state variables to represent its state during its lifetime. The mapping corresponding to the above is  $f^t$  where  $f^t(i) = |\{j : j \in \mathcal{A}^t, j \leq i\}|$  when  $i \in \mathcal{A}^t$  and  $f^t(i) = k + 1$  when  $T_i$  is attempting to leave the *start* state.

More than one transaction instance starting at the same time can be handled by marking out groups of instances which

become active together, but this is not considered in this paper.

#### D. State Space Traversal

The reachable state space is explored using a standard fixpoint computation. (In fixpoint computations which use sets of states, all the states must be expressed using the same fixed set of state variables. This was achieved by the dynamic mapping  $f^t$ .) Such a computation is valid when the property being checked depends at most on the relative ordering of transaction instance numbers (e.g.  $i < j$ ), and not on the precise value (e.g.  $i = 10$ ) or the separation (e.g.  $i = j + 3$ ) of the instance numbers.

The choice of the  $f^t$  described in Sec. IV-C can also be viewed as a means of breaking the symmetry in the state space. Consider the following two possibilities  $\mathcal{A}^t = \{4, 5, 6\}$  and  $\mathcal{A}^t = \{4, 5, 7\}$  where transaction instances  $T_6$  and  $T_7$  have the same value of associated state as do the other corresponding transaction instances  $T_4$  and  $T_5$ . Both of these map to the same assignment of the state variables  $S_1 \cup S_2 \cup S_3$ .

#### V. HAZARD CHECKING

In this section, we illustrate how a  $\mu$ -TLM with one transaction type  $T$  (and at most one new active transaction instance being created per cycle) can be checked for hazards using an automatable procedure. This illustrates how the transaction-level information which is easily accessible in the  $\mu$ -TLM supports automation of common verification tasks.

Any shared state which is written by  $T$  is a potential site for a read-after-write (RAW), write-after-read (WAR) or write-after-write (WAW) hazard. We show here how to check for the RAW hazard. (The WAR and WAW cases are similar.)

##### A. Checking RAW hazards

Assume that the number of active transactions  $|\mathcal{A}^t| \leq k$  at all times. Recall from Sec. IV-C that we encode the transient state using variables  $S_1 \cup S_2 \cup \dots \cup S_{k+1}$  such that transaction instance  $T_i$  uses the state variables  $S_{f^t(i)}$  at time  $t$ . The mapping  $f^t$  ensures that for a pair  $T_i$  and  $T_j$  of simultaneously active transaction instances,  $(i < j) \Rightarrow (f^t(i) < f^t(j))$ . Initially  $\mathcal{A}^0 = \{\}$  and  $f^0(1) = k + 1$  (the superscript refers to time). The initial state of the system has  $S_m^0 = end$  for  $1 \leq m \leq k$  and  $S_{k+1}^0 = start$ .

Suppose that  $r$  is a shared state element written to by transaction  $T$ . In order to check for a RAW hazard involving  $r$ , we introduce a new state element  $z_r$  with values  $1 \leq z_r \leq k + 1$ . Initially  $z_r^0 = k + 1$ . The value of  $z_r^t$  is always such that  $S_{z_r^t}$  is used by the most recently started (youngest) transaction instance to “effectively” have read from  $r$  before time  $t$ . For each  $m \in \{1, 2, \dots, k + 1\}$ , introduce signals  $writer_m$  and  $reader_m$ , where  $writer_m^t$  is true iff the transaction instance corresponding to  $S_m$  writes to  $r$  at time  $t$ , and similarly for  $reader_m^t$ . The state update for  $z_r^t$  involves two steps:

- 1) Identify the youngest transaction instance which has read from  $r$  before or at time  $t$ .

$$R^t = \{z_r^t\} \cup \{m : 1 \leq m \leq k + 1, reader_m^t \text{ is true}\}$$

$$y_r^t = \max(R^t)$$

- 2) If that youngest reader (associated with  $S_{y_r^t}$ ) is finishing execution at time  $t$ , then the next younger transaction instance which is alive at time  $t + 1$  inherits the role of the youngest reader. If all younger transaction instances are finishing execution at  $t$ , the readership is inherited by the possibly new transaction instance associated with  $S_{k+1}$  at time  $t + 1$ . This is needed since a finished transaction may be a potential reader for a RAW hazard whose write has yet to occur.

$$E^{t+1} = \{f^{t+1}(i) : f^t(i) \geq y_r^t, T_i \text{ alive at } t + 1\}$$

$$z_r^{t+1} = \min(E^{t+1} \cup \{k + 1\})$$

The absence of RAW hazards is now stated as the assertion  $G(\forall m, 1 \leq m \leq k + 1 : writer_m \Rightarrow y_r \leq m)$  where  $G$  is the ‘always’ operator of temporal logic. (This is the property  $r\_no\_RAW$ .) In addition, we have to verify that the bound  $|\mathcal{A}^t| \leq k$  is valid. (This is the property  $enough\_slots$ .) The properties  $r\_no\_RAW$  and  $enough\_slots$  together imply that the model does not suffer from a RAW hazard involving  $r$ . (As a sanity check, we also verify that at most one transaction instance writes to  $r$  at a time - the property  $r\_one\_write$ .)

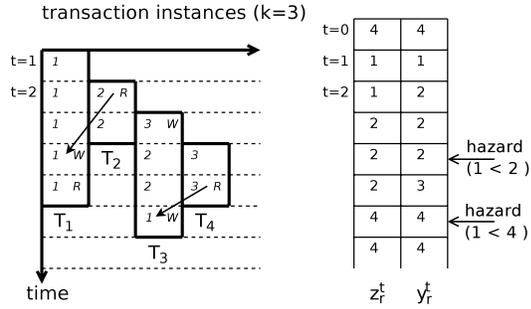


Fig. 2. Illustration of RAW hazard check,  $|\mathcal{A}^t| \leq 3$ .

Figure 2 above shows an execution trace for four transaction instances with two RAW hazards. Reads (Writes) to  $r$  are annotated with a R(W). Each transaction instance is also annotated with  $f^t(i)$  over its lifetime. The table on the right shows the values of  $z_r^t$  and  $y_r^t$  for this duration.

##### B. Automation of the RAW Hazard Check

The steps involved in model checking a  $\mu$ -TLM model with a single transaction type  $T$  for a RAW hazard are given below. They can be fully automated.

- Encode the system as described in Sec. IV-C assuming  $|\mathcal{A}^t| \leq k$ . Start with  $k = 1$  and increment  $k$  until the property  $enough\_slots$  becomes true.
- For each state element  $r$  written by  $T$ ,
  - Introduce the additional state  $z_r$  and write the equations for the update of  $z_r$  as in Sec. V-A
  - Verify the properties  $r\_no\_RAW$  and  $r\_one\_write$  using a model checker

### C. Case Study

Consider a simple  $\mu$ -TLM example with a single transaction type representing a simple pipeline with 4 stages *Read*, *Delay1*, *Write* and *Delay2*. The shared state consists of a single register file *Reg* with  $n$  registers and each register *Reg*[ $i$ ] is associated with a resource  $\rho_i$ . The local state maintained with each transaction instance is  $idx$  which takes values in  $\{0, 1, \dots, n-1\}$ . The transaction instance starts in stage *Read* with a non-deterministically chosen value for  $idx$ . The *Read* stage needs the resource  $\rho_{idx}$  to read *Reg*[ $idx$ ]. Once the resource  $\rho_{idx}$  is obtained, it is retained by the transaction instance until it finishes writing to *Reg*[ $idx$ ] in stage *Write*. The transaction instance does nothing in the *Delay1* and *Delay2* stages. Execution ends after the *Delay2* stage. We have currently hand-written the SMV input for the Cadence SMV model checker for this example. With  $n = 4$ , checking for the absence of RAW hazards takes 10 seconds ( $k = 1, 2, 3$ ) with the Linux version of Cadence SMV on a Pentium IV machine with 512 KB cache and 1GB RAM. The verification time grows rapidly with increasing  $n$  as the BDDs grow larger and the number of potential hazard sites increases. The example with  $n = 5$  takes 60 seconds to verify. A version of the pipeline model where the transaction does not retain  $\rho_{idx}$ , but requests it separately during the *Read* and *Write* steps, results in counterexamples being found in 1 second for  $n = 4$ , and 3 seconds for  $n = 5$ .

## VI. RELATED WORK

We are unaware of other work that directly model checks a  $\mu$ -TLM model for properties involving transaction ordering. The work related to model checking concurrent software (e.g. [2]) has some overlap as it shares some features of a dynamic state space, due to unbounded stack depth of recursion.

A lot of work related to checking for data hazards appears in the domain of processor verification. Here, the problem of verifying a  $\mu$ -architecture against a transaction-level (ISA) specification is well studied. There are many approaches for this using various correctness criteria (which imply absence of data hazards) and with varying degrees of automation. None of these approaches starts with a  $\mu$ -TLM model, and all these techniques need to manually make transaction (ISA) information accessible in the time-stationary model. The “flushing” approach [3] requires a way to limit the creation of new transaction instances (instructions). Compositional model checking (e.g. [4]) uses some auxiliary variables to define a refinement map between the architecture level and the RTL. Other related approaches (e.g. [5]–[8]) similarly require various aspects of transaction-level input. BlueSpec [9] uses operation-level, as opposed to a transaction level, modeling. It is unclear how hazard checking can be automated in that framework.

## VII. CONCLUSION

Traditional RTL based hardware verification presents a time-stationary view of the state space for property specification and verification using model checking. We argue that

this makes it difficult to directly state properties involving transaction information (e.g. properties which involve not just the temporal ordering of events, but also the relative ordering of transactions). For this purpose, we advocate the use of a transaction-level  $\mu$ -architecture model which describes computation on units of data in terms of individual transactions.

While  $\mu$ -TLM models make it easier to specify a richer set of properties, they pose a new challenge for model checking. Since transactions start and end, the state space seen by the model checker is dynamic. Our proposed solution to this encodes this dynamic state space using a fixed number of state variables while preserving transaction information, and then determines the fixed-point in the state space traversal. In the process, we use an assume-guarantee reasoning on the maximum number of in-flight transactions, and a dynamic transaction order preserving re-indexing of state variables.

With traditional RTL models, significant human input is needed to understand and augment the design so as to specify the hazards using only signals in the RTL design. With a  $\mu$ -TLM model, both specifying the properties which check for hazards and checking them are fully automatable. We show an implementation of these ideas in SMV for a simple pipeline.

Transaction-level modeling has been pushed as a path to greater designer productivity due to its higher abstraction level; in this paper we show that through appropriate augmenting of model checking algorithms, it can also be made useful for greater automation in data-level property specification and verification.

## ACKNOWLEDGEMENT

The authors acknowledge the support of the Gigascale System Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

## REFERENCES

- [1] Y. Mahajan, C. Chan, A. Bayazit, S. Malik, and W. Qin, “Verification driven formal architecture and microarchitecture modeling,” in *MEM-OCODE*, 2007.
- [2] S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in *TACAS*, 2005.
- [3] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessor control,” in *Proceedings of the Sixth International Conference on Computer-Aided Verification CAV*, David L. Dill, Ed., vol. 818. Stanford, California, USA: Springer-Verlag, 1994, pp. 68–80.
- [4] R. Jhala and K. L. McMillan, “Microarchitecture verification by compositional model checking,” in *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2001, pp. 396–410.
- [5] J. Sawada and W. A. Hunt, “Processor verification with precise exceptions and speculative execution,” in *Proc. 10th International Computer Aided Verification Conference*, 1998, pp. 135–146.
- [6] R. Hosabettu, G. Gopalakrishnan, and M. Srivas, “Formal verification of a complex pipelined processor,” *Form. Methods Syst. Des.*, vol. 23, no. 2, pp. 171–213, 2003.
- [7] S. Lahiri, S. Seshia, and R. Bryant, “Modeling and verification of out-of-order microprocessors in UCLID,” in *(FMCAD '02), LNCS 2517*. Springer-Verlag, 2002, pp. 142–159.
- [8] J. T. Higgins and M. D. Aagaard, “Simplifying the design and automating the verification of pipelines with structural hazards,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, no. 4, pp. 651–672, 2005.
- [9] Arvind and X. Shen, “Using term rewriting systems to design and verify processors,” *IEEE Micro*, vol. 19, no. 3, pp. 36–46, 1999.

# Computing Predicate Abstractions by Integrating BDDs and SMT Solvers

R. Cavada    A. Cimatti    A. Franzén    K. Kalyanasundaram    M. Roveri    R. K. Shyamasundar  
 FBK-irst    FBK-irst    FBK-irst    TIFR-Mumbai & FBK-irst    FBK-irst    TIFR-Mumbai  
 cavada@itc.it    cimatti@itc.it    franzen@itc.it    krishnamani@itc.it    roveri@itc.it    shyam@acm.org

## Abstract—

The efficient computation of exact abstractions of a concrete program for a given set of predicates is key to the efficiency of Counter-Example Guided Abstraction-Refinement (CEGAR). Recent work propose the use of DPLL-based SMT solvers, modified into enumerators. This technique has been successfully applied in the realm of software, where a control flow graph is available to direct the exploration. However this approach shows some limitations when the number of models grows: in fact, it intrinsically relies on the enumeration of all the implicants, which basically requires the enumerations of all the disjuncts in the DNF of the abstraction.

In this paper, we propose a new technique to improve the construction of abstractions. We complement SMT solvers with the use of BDDs, which enables us to avoid the model explosion. Essentially, we exploit the fact that BDDs are a DAG representations of the space that a DPLL-based enumerator treats as a tree. A preliminary experimental evaluation shows the potential of the approach.

## I. INTRODUCTION

Abstraction is one of the key techniques to tackle the state-space explosion problem. Instead of directly searching the system under analysis (also called the concrete system), information is disregarded in order to obtain an abstract system, which is easier to analyze. The analysis of the abstract system is intended to provide information useful to evaluate the properties of the concrete system. However, since the abstraction may throw away important information, the result of the analysis in the abstract space may not be conclusive, and additional refinement steps may be in order. A popular framework is *Counter-Example Guided Abstraction Refinement* (CEGAR) [1]. CEGAR uses conservative abstractions, i.e. each trace in the concrete space has a counterpart in the abstract space. This means that, in the case of invariant properties, if the analysis in the abstract space reveals no bugs, then the concrete system is also correct. When an abstract counterexample is found, concretization is attempted, i.e. a correspondent counterexample in the concrete system is searched. If it cannot be found, then the abstract counterexample is deemed spurious, and the reasons for failure to concretize are analyzed, in order to decide which information should be added to the abstraction. The CEGAR approach has seen substantial success in recent years [2]. This technique is now being used to do compositional verification [3], to verify device drivers, ANSI-C programs [4], [5], RTL designs written in Verilog HDL [6], SpecC [7], etc.

An efficient way to obtain a conservative abstraction is localization reduction, where certain signals are simply replaced with inputs. However, the resulting abstraction can be very coarse, since the correlation between abstracted signals is completely lost. A more informed approach is *predicate abstraction*: the concrete state space is partitioned by a set of predicates; each predicate is associated with a boolean variable, so that each part of the concrete space corresponds to an abstract state; the abstract transition relation is defined accordingly. However, the exact computation of the abstract system given a set of predicates is a potential bottleneck, and [6], [8] investigate the computation of approximate abstractions. The downside is that approximating the abstraction may yield more spurious counterexamples, and result in additional CEGAR iterations.

The exact computation of abstractions is tackled in [9] by extending a DPLL-based Satisfiability Modulo Theory (SMT) solver to work as a model enumerator. This technique has been successfully applied in the realm of software, where a control flow graph directs the exploration of the abstract space, and the emphasis is on the solution of constraints in the background theory. However, this approach is subject to the model explosion problem: it has to enumerate enough implicants to cover the abstraction, and this boils down to going through the construction of the corresponding DNF.

In this paper, we propose a new technique for the computation of exact predicate abstractions, that is more efficient in dealing with the boolean components. Our approach integrates Binary Decision Diagrams (BDDs) [10] and procedures for Satisfiability Modulo Theory (SMT) [11]. It is based on the consideration that BDD-based quantification methods are highly optimized, and can be vastly superior to DPLL-based enumerators, especially within a certain (reasonable) number of variables. Our method applies a modified quantification on (the BDD representing) the boolean abstraction of the problem, that interacts with an SMT solver in order to ensure the consistency of the models with respect to the background theory. The traversal of the BDD recursively computes the results of quantifications for the subtrees, and combines them. In addition, the theory constraints that are “activated” on the current path are sent for consistency checking to an SMT solver working as an oracle on the side. Essentially, we exploit the fact that BDDs are a DAG representation of the space that a DPLL-based enumerator treats as a tree. The method

retains the advantages of different forms of caching in BDDs (where models are accumulated) and in SMT (where theory conflicts are discovered and stored). It is compatible with advanced BDD-based techniques, and nicely integrates within the CEGAR loop, providing incrementality in the computation of the abstraction. The experimental evaluation shows that our algorithm is able to outperform state-of-the-art existential quantification based on SMT.

This paper is structured as follows. In Section II we outline some background. In Section III we propose the SMT-framework for CEGAR. In Section IV we discuss the BDD-SMT abstraction algorithm, while in Section V we discuss the details of the implementation. In Section VI we compare our approach to related work, and in Section VII we carry out an experimental analysis. In Section VIII we draw some conclusions and outline future work.

## II. TECHNICAL PRELIMINARIES

Our setting is standard first order logic. We consider a signature to be composed by individual constants and variables, function symbols, boolean variables, and predicate symbols. A term is either a constant, a variable, or the application of a function symbol of arity  $n$  to  $n$  terms. A theory constraint (also called a theory atom) is the application of a predicate symbol of arity  $n$  to  $n$  terms. An atom is either a theory constraint or a boolean variable. A literal is either an atom or its negation. A formula is either true ( $\top$ ) or false ( $\perp$ ), a boolean variable, a theory constraint, the application of a propositional connective of arity  $n$  to  $n$  formulae, or the application of a quantifier to an individual variable and a formula. We use  $x, x', x_1, x_2, \dots, y, y', y_1, y_2, \dots$  for individual variables, and  $\vec{x}, \vec{y}, \vec{v}, \dots$  for vectors of individual variables. Terms and formulae are referred to as expressions, denoted with  $e, e', e_1, e_2, \dots$ . We use  $c, c_1, c_2, \dots$  for theory constraints,  $c(x)$  to stress the dependence of  $c$  on  $x$ , and  $c(\vec{x})$  to stress the dependence on  $\vec{x}$ . We use  $P, P', P_1, P_2, \dots, Q, Q', Q_1, Q_2, \dots$  for boolean variables, and  $\vec{P}, \vec{Q}, \dots$  for vectors of boolean variables; we write  $\vec{P}_i$  for the  $i$ -th variable in  $\vec{P}$ . We write  $\Phi(Q)$  to highlight the fact that  $Q$  occurs in  $\Phi$ .

Substitution is defined in the standard way (see for instance [12]). We write  $\phi[e/e']$  for the substitution of every occurrence of  $e$  in  $\phi$  with  $e'$ . If  $\vec{e}$  and  $\vec{e}'$  are vectors of (either individual or boolean) expressions of the same size, we write  $\phi[\vec{e}/\vec{e}']$  for the parallel substitution of every occurrence of  $\vec{e}_i$  (the  $i$ -th element of  $\vec{e}$ ) in  $\phi$  with  $\vec{e}'_i$  (the  $i$ -th element of  $\vec{e}'$ ). We use boolean quantification (i.e. quantification over boolean variables)  $\exists Q.(\Phi(Q))$  as a shortcut for  $\Phi[Q/\top] \vee \Phi[Q/\perp]$ .

We use the standard semantic notion of interpretation and satisfiability. We call truth assignment for a set (vector) of atoms  $\vec{Q}$  a total function  $\mu : \vec{Q} \rightarrow \{\top, \perp\}$ . In propositional logic, checking the satisfiability (SAT) of a formula is finding a truth assignment to the variables that will make it evaluate to true. The problem is approached with enhancements of the DPLL algorithm: the formula is converted into an equisatisfiable one in Conjunctive Normal Form (CNF); then, a truth assignment is incrementally built, until either all the clauses

are satisfied, or a conflict is found, in which case backjumping takes places (i.e. certain assignments are undone). Keys to efficiency are heuristics for the variable selection, and learning of conflicts (see e.g. [13]). Reduced Ordered Binary Decision Diagrams (in the following simply called BDDs) [10] are a canonical representation for propositional logic. Given a total order for the propositional variables, there exists a unique equivalent BDD, that represents as a DAG all the satisfying assignments. Efficient packages are available [14] to construct the BDDs corresponding to a formula; operations are also available to compute the BDD corresponding to quantified boolean formulae. SAT solvers and BDD packages are backbones to many formal verification tools.

The more general problem of SMT (satisfiability modulo a background theory) corresponds to deciding whether there exists an interpretation in the theory that satisfies the formula. Examples of useful theories are equality and uninterpreted functions (EUF), difference logic (DL) linear arithmetic (LA), either over reals or the integers, the theory of bit vectors (BV), and their combination.

We call the boolean abstraction of  $\phi$  the formula  $\Phi[\vec{c}/\vec{Q}]$ , where  $\vec{c}$  is the vector of constraints occurring in  $\phi$ , and  $\vec{Q}$  is a set of boolean variables not occurring in  $\Phi$ . Let  $\mu : \vec{Q} \rightarrow \{\top, \perp\}$  be a truth assignment to the variables in  $\vec{Q}$ . The set of constraints induced by  $\mu$  is defined as  $\{c_i(\vec{x}) \mid \mu(Q_i) = \top\} \cup \{\neg c_i(\vec{x}) \mid \mu(Q_i) = \perp\}$ . We say that  $\mu$  is theory-consistent iff so is the corresponding set of constraints. Most SMT solvers are based on the enumeration of the models of the boolean abstraction, combined with a check of theory-consistency. Typically, the DPLL algorithm is modified to enumerate all models, and a number of optimizations are applied in order to improve efficiency. Among these, we mention: *early pruning* (i.e. applying a theory consistency check on the partial model, so that theory inconsistent models are not explored further); construction of *theory conflict clause* (i.e. detecting a small subset of the theory constraints in the inconsistent assignment), to be used for conflict analysis, backjumping, and learning; *theory deduction* (i.e. for constraints that are currently unassigned, detection of truth values implied by the other theory constraints). See [11] for a thorough discussion.

## III. THE CEGAR FRAMEWORK

We consider symbolically represented systems. Given a theory, we consider a vector of current state (either individual or boolean) variables  $\vec{v}$ , and a vector of next-state variables  $\vec{v}'$ . A state is an assignment to a vector of state (either individual or boolean) variables  $\vec{v}$ , and a transition is a pair of assignments to current and next state variables. A formula  $\phi(\vec{v})$  represents a set of states (e.g. the initial states, or a property) while  $\Psi(\vec{v}, \vec{v}')$  represents a set of transitions, where  $\vec{v}'$  is the vector of next state variables. Let us consider a system with initial states  $\mathcal{I}_c$  and transition  $\mathcal{R}_c$ , and let  $\mathcal{P}_c$  be a property to be verified.

The *Counterexample Guided Abstraction Refinement (CEGAR) framework* [1] framework is shown in Fig. 1. Instead of

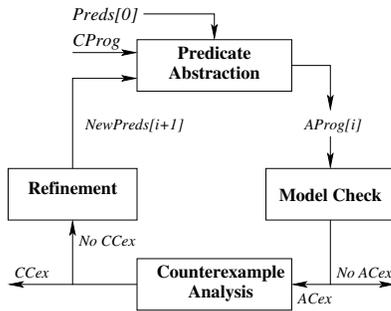


Fig. 1. The CEGAR Framework

searching the state space of the concrete program directly, we create an abstraction of the program, which is amenable for model checking. The abstraction is typically constructed to be conservative, that is, every trace in the concrete space has a counterpart in the abstract space. If there are no bugs in the abstract space, then there are no bugs in the original system. However, if an abstract counterexample exists, there may not be a corresponding counterexample for the concrete system. Such an abstract counterexample is then called a *spurious counterexample*. Then, abstraction-refinement tries to discover a new abstract model, which contains more detail in order to rule out spurious counterexamples. This is done by extracting information from counterexamples generated by the model checker. The process is iterated until the property is either proved or disproved.

We focus on *Predicate Abstraction* [15], one of the most widely used abstraction techniques, where the abstract program is constructed based on a given set of predicates  $\gamma_i(\vec{v})$  on the variables of the concrete program. Each such predicate is represented by a boolean variable  $P_i$  in the abstract program. Intuitively, each abstract variable  $P_i$  partitions the concrete space in two sets: one is the set of states satisfying  $\gamma_i$ , associated to the case when  $P_i$  is true, and the other one is the complement ( $P_i$  false). Thus, an abstract state is associated with the set of concrete states corresponding to the intersection of the sets corresponding to each abstract literal. The transition relation of the abstract program is such that two abstract states are related iff there exist two concrete states in the respective concretizations that are in the concrete transition relation.

The initial states  $\mathcal{I}_a(\vec{P})$ , the transition relation  $\mathcal{R}_a(\vec{P}, \vec{P}')$  and the property  $\mathcal{P}_a(\vec{P})$  of the abstract program are defined using *existential abstraction*. We find out states (transitions among the states) in the abstract program such that *there exists* corresponding states (transitions) in the concrete program involving concrete variables. More formally,

$$\mathcal{I}_a(\vec{P}) \doteq \exists \vec{v}. (\mathcal{I}_c(\vec{v}) \wedge \bigwedge_i P_i \leftrightarrow \gamma_i(\vec{v})) \quad (1)$$

$$\mathcal{R}_a(\vec{P}, \vec{P}') \doteq \exists \vec{v}, \vec{v}'. (\mathcal{R}_c(\vec{v}, \vec{v}') \wedge \bigwedge_i P_i \leftrightarrow \gamma_i(\vec{v}) \wedge \bigwedge_i P'_i \leftrightarrow \gamma_i(\vec{v}')) \quad (2)$$

$$\mathcal{P}_a(\vec{P}) \doteq \exists \vec{v}. (\mathcal{P}_c(\vec{v}) \wedge \bigwedge_i P_i \leftrightarrow \gamma_i(\vec{v})) \quad (3)$$

The abstract system thus obtained is purely boolean, and can be subjected to model checking to verify the abstraction of the properties of interest.

Let us now consider a sequence of abstract states  $s_{[0]}, \dots, s_{[k]}$  (where each abstract state  $s_{[i]}$  is a valuation for  $\vec{P}$ ). If the abstract property does not hold in the abstract model we generate an abstract counterexample that must be checked for spuriousness, i.e. we check whether it can be refined in the concrete space. This can be done with a setting similar to bounded model checking, where each state vector of both the concrete and abstract machine are replicated at different time steps, from 0 to  $k$ . We write  $\vec{v}_{[h]}$  for the replica of the concrete state vector at time  $h$ ,  $\vec{P}_{[h]}$  for the  $h$ -th abstract state vector replica, and  $\vec{\gamma}(\vec{v}_{[h]})$  for the vector of predicates instantiated at time  $h$ . Checking the spuriousness of a counterexample corresponds to checking the satisfiability of the following formula:

$$\mathcal{I}_c(\vec{v}_{[0]}) \wedge \bigwedge_{h=0}^{k-1} \mathcal{R}_c(\vec{v}_{[h]}, \vec{v}_{[h+1]}) \wedge \bigwedge_{h=0}^k \vec{P}_{[h]} \leftrightarrow \vec{\gamma}(\vec{v}_{[h]}) \wedge \neg \mathcal{P}_c(\vec{v}_{[k]})$$

The main idea behind the refinement phase is to learn more information from the spurious counterexamples produced and use the information to refine the abstraction in such a way that it rules out the spurious counterexample. Spurious transitions are those abstract transitions that do not have any corresponding concrete transitions. If the most precise abstraction with respect to the given set of predicates is computed, the spuriousness of the counterexample would be because of insufficient number of predicates, i.e. the absence of information rich enough to capture all the relevant behaviours of the concrete system, even for the most precise abstraction. In the rest of this paper, we focus on the phase of the CEGAR loop concerned with the construction of the abstraction.

#### IV. ABSTRACTION BY BDDs MODULO THEORY

We describe the algorithm required for generation of the abstract system in a more general setting, using the following notation:  $\Phi(\vec{x})$  is the formula to be abstracted;  $\vec{x}$  is the set of variables to be abstracted (notice that they can include both individual and boolean variables, both current state and next state).  $\vec{V}$  is the set of boolean variables to be retained, also called important variables. We want to compute a propositional formula equivalent to

$$\exists \vec{x}. (\Phi(\vec{x}) \wedge \bigwedge_i V_i \leftrightarrow \gamma_i(\vec{x}))$$

In the following, let  $\vec{c}$  be the vector of theory constraints occurring either in  $\Phi$  or in some  $\gamma_i$ . Our approach is based on the following steps. First, for each theory constraint  $c_i(\vec{x})$  in  $\vec{c}$  we create a fresh boolean (BDD) variable  $Q_i$ , called its boolean abstraction, we group all the  $Q_i$  in  $\vec{Q}$ , and the abstraction bijection  $\mathcal{A}$  is defined as  $\{\{Q_i \cdot c_i(\vec{x})\}\}$ . Then, we build  $(\Phi(\vec{x}) \wedge \bigwedge_i V_i \leftrightarrow \gamma_i(\vec{x}))[\vec{c}/\vec{Q}]$ , i.e. the boolean formula in the  $\vec{V}$  and  $\vec{Q}$ , by replacing each theory atom  $c_i(\vec{x})$  with the corresponding boolean abstraction  $Q_i$  from the matrix of the

```

1: function BddThAbstract( $b, C, V$ )
2:   if ( $b = \top$ )  $\vee$  ( $b = \perp$ ) then return  $b$ 
3:    $v := \text{var}(b)$ ;
4:   if BooleanAtom( $v$ ) then
5:      $tt := \text{BddThAbstract}(\text{BddThen}(b), C, V)$ 
6:      $ee := \text{BddThAbstract}(\text{BddElse}(b), C, V)$ 
7:     if  $v \in V$  then
8:       return BddITE( $v, tt, ee$ )
9:     else
10:      return BddOr( $tt, ee$ )
11:   else
12:      $c_v := \text{VarToConstraint}(v)$ ;
13:     if BddThen( $b$ ) =  $\perp$  or ThInconsistent( $C, c_v$ ) then
14:        $tt := \perp$ 
15:     else
16:        $tt := \text{BddThAbstract}(\text{BddThen}(b), C \cup \{c_v\}, V)$ 
17:     if BddElse( $b$ ) =  $\perp$  or ThInconsistent( $C, \neg c_v$ ) then
18:        $ee := \perp$ 
19:     else
20:        $ee := \text{BddThAbstract}(\text{BddElse}(b), C \cup \{\neg c_v\}, V)$ 
21:     return BddOr( $tt, ee$ )
22: end function

```

Fig. 2. The function for existential quantification Modulo Theories

above formula. With standard techniques, we then construct the corresponding BDD representation, that we also denote as  $\Psi(\vec{V}, \vec{Q})$ .

If we apply a quantification of the unimportant variables  $\vec{Q}$ , we are left with a formula in the  $\vec{V}$  variables. However, this is an overapproximation of the desired result, since the relation between theory constraints is completely lost. We would obtain the correct result if we were able to obtain  $\text{PruneMT}(T, \Psi(\vec{V}, \vec{Q}))$ , i.e. a simplification of the BDD  $\Psi(\vec{V}, \vec{Q})$ , so that only the paths that are consistent with respect to the background theory are left.  $\text{PruneMT}(T, \Psi(\vec{V}, \vec{Q}))$  propositionally implies  $\Psi(\vec{V}, \vec{Q})$ , and must be satisfied by every  $T$ -consistent model of  $\Psi(\vec{V}, \vec{Q})$ . In a sense,  $\text{PruneMT}(T, \Psi(\vec{V}, \vec{Q}))$  is the lifting to the boolean case of the relationships between theory constraints. A naive implementation of pruning modulo theories could be to enumerate the satisfying assignments of the BDD. If one is found to be theory-inconsistent, then we negate it (or perhaps the conflict set), we conjoin it with the original formula, and we restart. The resulting BDD is such that all the assignments are also theory consistent. We proceed until we reach a fix point. For each truth assignment, a call to the theory solver is performed. Unfortunately, such approach is likely to be extremely inefficient, for two reasons. First, producing  $\text{PruneMT}(T, \Psi(\vec{V}, \vec{Q}))$  may generate overly big BDDs, either in the intermediate computations, or as a final result. Second, the approach is in a sense eager, in that it may “lift” to the boolean level parts of the theory that are not required, because of the subsequent quantification.

The approach we propose, depicted in Fig. 2, interleaves within the same routine boolean quantification and pruning modulo theory. We solve our problem by calling `BddThAbstract` with arguments  $\Phi, \{ \}, \vec{V}$ . The  $\mathcal{A}$  mapping between theory constraints and their boolean variables is assumed to be globally available. The function is recursively

defined over the structure of the BDD corresponding to the first argument, and representing the first formula to be quantified. The second argument is a set of constraints, called the context of simplification, while the third one is the set of important variables.

The algorithm is best explained as an extension of the existential quantification in the purely boolean case, i.e. when the  $\mathcal{A}$  mapping is empty, so that the `BooleanAtom` test always returns true. The lines from 11 to 21 are never executed, and the algorithm boils down to standard existential quantification for BDDs. In the base case, if  $b$  is either  $\top$  or  $\perp$ , then it is simply returned (line 2). Otherwise, recursive calls are applied both to the then and the else branches (lines 5 and 6). If the variable  $v$  is important, then it must not be quantified, and the results of the recursive calls are combined into an if-then-else node, otherwise they are disjoined (lines 7-10).

In the more interesting case where some variables are indeed theory constraints, then we apply a form of pruning, that attempts to ensure theory-consistency of the traversed paths. The key idea is the simplification context, i.e. the set of theory constraints that “get activated” when descending from the root to the node  $b$ . If the current variable is the abstraction of a theory atom, then the current context can be extended with either a positive or negative constraint, depending on the branch we expand first. However, in order for either expansion to lead to a model, it has to be theory consistent: if either extension is inconsistent with the current context, then the evaluation can be safely pruned. If the context extended with the positive constraint is not satisfiable (line 13), then we simply assign  $\perp$  for the right branch  $tt$ . In fact, there is no way the path can be extended to a theory-consistent assignment. In case of consistency, we recur with extended context  $C \cup \{c_v\}$ . Similar approach is taken for the left branch. The results are recombined with disjunction, since the boolean abstraction of constraints are quantified out.

The algorithm can also be interpreted in the setting of SMT solving. The BDD can be thought of as a compact representation of the boolean search space. The traversal of the BDD can be seen as an enumerator, that differs from the standard DPLL-approach in two ways. First, the order of traversal is fixed (which is potentially a drawback). Second, the BDD treats the search space as a DAG, while DPLL-based enumerators are in fact traversing the corresponding tree; learning can be seen as an attempt to mitigate this problem. Checking the consistency of the context extension is the counterpart of *early pruning* in DPLL-based SMT solvers.

## V. IMPLEMENTATION AND INTEGRATION IN CEGAR

From the practical view point, the implementation of the algorithm is not trivial, due to the combination of a BDD package and an SMT solver – delicate balances have to be taken into account.

To deal with the theory part, we use a Theory Context Checker (TCC), which can be thought of as an SMT solver, where the top level has been stripped out: it has a stack, where

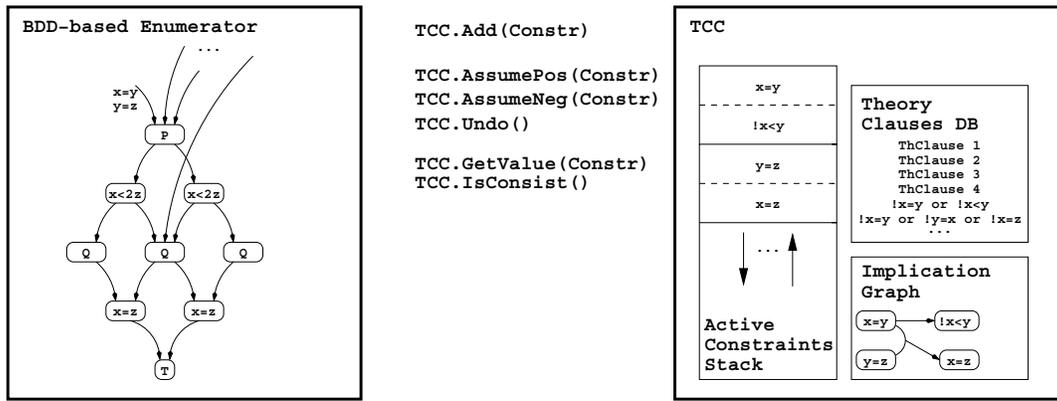


Fig. 3. The interaction between the BDDs and the SMT engine

constraints can be activated, provides conflict detection and storage, and can suggest values for variables.

The BDD enumerator and the TCC interact according to the interface depicted in Fig 3. The TCC status, corresponding to the active constraints, is managed in a stack-based manner. The enumerator can ask the TCC to extend it assuming a constraint positive or negative, or to undo the last assumption. The enumerator can ask the TCC whether its status is consistent, and ask for the current value of a constraint. Depending on the theory, consistency checking can be carried out with an incomplete (possibly cheaper) procedure; in such case it is important to carry out a complete check when a complete model is found.

Some remarks are in order. First, the TCC detects reasons for inconsistencies (conflict sets), stores them as conflict clauses, and carries out conflict analysis. In case of inconsistency, it can tell the enumerator the point of backtracking necessary to undo the inconsistency. Obviously, in the case we carry out early pruning with a complete procedure, we are guaranteed that the status was consistent until the latest addition, and thus only one step needs to be undone.

Second, it can carry out theory deductions, and boolean constraint propagations over theory clauses. It is thus possible that a constraint that is unassigned in the current BDD path must in fact have a value. Third, the role of boolean constraint propagation is to avoid repeating the same theory checks: constraints that obtain a value because of Boolean Constraint Propagation (BCP) are guaranteed to be theory consistent.

As for constraints unassigned in the BDD enumerator, with values implied by the TCC status, there are two ways to proceed. The choice that we currently implement is to delay taking them into account until the corresponding level is reached in the enumeration. Another possibility would be to simplify the remaining BDD according to the corresponding value before continuing with enumeration.

Most of the tricks in the BDD package (e.g. constant time negation based on pointer complementation) can be retained without changes. However, this is not the case for *memoization*. Memoization plays a very important role in BDD

processing: it can be thought of as a way to avoid processing a BDD (which is a DAG) as if it were a tree. In fact, the first time we compute the result of an operation (e.g. the quantification of node  $b$  results in  $\top$ ), we store it in a hash table, and so that when the node is visited again (as a consequence of descending a BDD along a different path) it can be simply retrieved and returned. Unfortunately, this mechanism can not be applied in a straightforward manner in our case, because of the dependence on the context of the result of quantification: if the same node is reached following two different paths that activate different sets of theory constraints, the result of the call is not the same in the general case. There are several approaches to this problem. The first is simply to disregard memoization. The second one is to memoize based on the dependence of the context. The third one is to exploit subsumption between contexts: if the quantification of node  $b$  under context  $C$  has been computed to be  $r$ , then we can know that the quantification of  $b$  under context  $C^- \subset C$  is implied by  $r$ ; similarly, the quantification of  $b$  under  $C^+ \supset C$  implies  $r$ . This information may be particularly valuable in the cases of  $C^-$  when  $r$  is  $\perp$ , and in the case of  $C^+$  when the result is  $\top$ . However, the tradeoff between the cost of checking context subsumption and the value of the information has to be investigated. We currently take the first approach, also on the grounds that some memoization is carried out in the recursive calls to disjunction on unimportant variables; a study of the trade-offs associated to the other solutions is object of future work.

We notice that the search can be enhanced by exploiting the levels of the BDD. In particular, if we see that a certain node is below the level of the last important variable, and the quantification of one of his children is not  $\perp$ , then there is no need to evaluate the other child.

We now discuss the integration of our approach in a CEGAR setting. We first notice that our approach is fully amenable to incrementality: The status of the TCC (i.e. the conflicts learned during the abstraction in the first iteration) can be used in the forthcoming iterations. Second, after one iteration, the constraints between the variables of the abstract space can

be used to constrain the enumeration at the boolean level; this is in a sense similar to what static learning [16] does.

In addition, the TCC clauses that lift the relationships between theory atoms to the boolean level, can be used to constrain reachability analysis in the abstract space, thus possibly obtaining a tighter characterization of the abstract space.

Finally, such invariants can be used in the step of refinement of the counterexample, in particular by shifting over time.

## VI. RELATED WORK

In this section, we compare our approach with three streams of work: existential quantification; computation of abstractions; integration between BDDs and SMT.

*Existential Quantification:* In the boolean setting, quantification is typically used for the basic operation of Symbolic Model Checking, i.e. image computation. BDD-based procedures have been optimized over the years [17]. More recently, SAT-based quantifier elimination has been investigated [18], [19]. Basically, once a satisfying assignment is found, the subset concerning the important variables is accumulated, and used as a blocking clause. Cube enumeration approaches suffer from a model explosion problem: the number of iterations is linear in the number of cubes - which can easily grow exponential. The most striking example is a set of clauses  $c^i = l_1^i, \dots, l_n^i$  for  $i = 1, \dots, m$ . When each clause has independent variables, the number of prime implicants is  $m^n$ , which is the number of iterations required. The work on cofactoring by Gupta et al. seems to provide a substantial advantage over cube-based enumeration [20]. However, it is unclear how this can be exploited in the setting of predicate abstraction, given the specific structure of the predicates to be retained. The work in [21] combines BDD-based and SAT-based techniques.

*Decision procedures for Abstraction:* The idea of using decision procedures for computing abstractions has been explored in [22]–[24]. The work in [9] improves over them by lifting DPLL-based quantification to the case of SMT. It also inherits the model explosion problem. In fact, both Barcelogic and our implementation in MathSAT of [9] suffer from exponential degrade in performance simply with the addition of clauses over fresh boolean variables.

A possible approach is to trade precision for accuracy. In fact, [9] also shows how to approximate the results. A similar line is followed in [6], [8], where different approximate methods for the computation of predicate abstractions are presented. The main problem is that approximation in the abstraction may lead to additional iterations in the CEGAR loop. In this paper, we concentrate on the computation of the exact abstraction for a given set of predicates.

*BDDs for SMT:* We are not aware of any other work integrating BDDs and SMT for quantification. The first versions of Harvey [25] uses BDDs as model enumerators; however, in solving “single” satisfiability modulo theory problems, DPLL-based enumeration techniques seem to be vastly superior. A BDD-based approach basically finds all the boolean models

before starting theory reasoning, which is basically overkill: we stop as soon as one model is found, and many boolean models may be theory inconsistent. Our choice of BDDs is motivated by the fact that all the models have to be explored, and also by the role played by quantification.

Armando [26] addresses a different problem, i.e. simplifying the boolean part of a theory formula while preserving equivalence; similar considerations apply to [27]. Closer to our approach is [28], that presents simplification of a BDD with respect to a background theory; the work is however limited to the theory of abstract data types, and does not deal with quantification.

Finally, BDD-based enumeration in an SMT setting is used for LTL satisfiability [29]. Main differences are that prime implicants are enumerated (rather than paths). Furthermore, in [29] there is no early pruning, and each prime implicant is managed separately, without exploiting the DAG structure of the BDD.

## VII. EXPERIMENTS

We tested the proposed algorithm within the NuSMT system, an implementation of the CEGAR loop integrating SMT techniques (specifically, the MathSAT solver) and the NuSMV model checker. NuSMT uses the NuSMV language extended to deal with real-valued and integer-valued variables. This allows us to present the concrete program with formulae characterizing the set of initial states, the invariants, and the transition relation. The abstract program is a finite state program. Although NuSMT implements a full-blown CEGAR loop, with counterexample refinement and predicate discovery, in the rest of this paper we focus on the predicate abstraction part.

In addition to the method presented in the previous section, we have also implemented in MathSAT (for a wide range of theories) the SMT-based method proposed in [9]. In the following we use AllMathSAT to refer to our implementation of the SMT-based method proposed in [9], and BDD+SMT to refer to the implementation of BddThAbstract. We conducted experiments on networks of hybrid automata of different sizes, with constraints in linear arithmetic over the reals. We compared our BDD+SMT abstractor with AllMathSAT. The benchmarks are available at <http://sra.itsc.it/people/roveri/fmcd07>. In the comparison, we were unable to include Barcelogic [9] for technical reasons related to the lack of expressivity required to deal with the benchmarks addressed in this paper. However, we believe that AllMathSAT can be considered as reasonably close to Barcelogic (and both show substantial degrade in performance when the problem has a high number of implicants).

A test case with name “*hann-ss-tt-vv*” abstracts a composition of  $n$  hybrid automata, each of them having  $s$  locations and  $t$  transitions. Each state of the automata is associated with an invariant, while each transition has both a precondition and an effect; all of them are formulae in LA over  $v$  variables.

All the experiments were run on a 3GHz Intel(TM) Xeon(TM) Dual Processor running Linux equipped with 4GB of RAM. For each experiment we fixed a memory limit of

Benchmark	BDD+SMT	AllMathSAT
han2-s5-t10-v5_9	.401	5.104
han2-s5-t10-v5_8	.334	4.988
han2-s5-t10-v5_7	.105	10.902
han2-s5-t10-v5_6	.181	6.152
han2-s5-t10-v5_5	.225	4.933
han2-s5-t10-v5_4	.137	5.903
han2-s5-t10-v5_3	.215	7.587
han2-s5-t10-v5_2	.421	4.973
han2-s5-t10-v5_1	.350	9.270
han2-s5-t10-v5_0	.362	6.531
han2-s5-t30-v5_9	.165	6.576
han2-s5-t30-v5_8	.224	5.844
han2-s5-t30-v5_7	.267	5.066
han2-s5-t30-v5_6	.932	8.566
han2-s5-t30-v5_5	.192	3.701
han2-s5-t30-v5_4	.416	3.952
han2-s5-t30-v5_3	1.577	8.667
han2-s5-t30-v5_2	1.439	4.045
han2-s5-t30-v5_1	.191	6.344
han2-s5-t30-v5_0	.203	5.597

TABLE I  
HYBRID AUTOMATA: A=2, S=5, T=10,30

Benchmark	BDD+SMT	AllMathSAT
han3-s5-t10-v5_9	1.579	573.552
han3-s5-t10-v5_8	1.904	610.045
han3-s5-t10-v5_7	3.152	661.323
han3-s5-t10-v5_6	5.422	644.350
han3-s5-t10-v5_5	3.813	747.640
han3-s5-t10-v5_4	2.170	496.716
han3-s5-t10-v5_3	2.620	597.161
han3-s5-t10-v5_2	2.247	554.023
han3-s5-t10-v5_1	1.833	329.028
han3-s5-t10-v5_0	2.965	585.501
han3-s5-t30-v5_9	19.840	575.759
han3-s5-t30-v5_8	12.318	485.817
han3-s5-t30-v5_7	42.894	480.681
han3-s5-t30-v5_6	189.370	591.740
han3-s5-t30-v5_5	131.216	462.630
han3-s5-t30-v5_4	185.886	681.669
han3-s5-t30-v5_3	131.115	502.559
han3-s5-t30-v5_2	56.284	453.767
han3-s5-t30-v5_1	164.576	619.993
han3-s5-t30-v5_0	65.202	672.153

TABLE III  
HYBRID AUTOMATA: A=3, S=5, T=10,30

Benchmark	BDD+SMT	AllMathSAT
han2-s10-t10-v5_9	.430	257.400
han2-s10-t10-v5_8	.419	268.184
han2-s10-t10-v5_7	.459	270.294
han2-s10-t10-v5_6	.375	320.186
han2-s10-t10-v5_5	.420	210.345
han2-s10-t10-v5_4	.314	274.153
han2-s10-t10-v5_3	.446	329.685
han2-s10-t10-v5_2	.284	221.098
han2-s10-t10-v5_1	.316	306.356
han2-s10-t10-v5_0	.462	319.609
han2-s10-t30-v5_9	5.396	268.046
han2-s10-t30-v5_8	5.050	283.478
han2-s10-t30-v5_7	4.969	352.261
han2-s10-t30-v5_6	4.157	362.505
han2-s10-t30-v5_5	5.480	328.484
han2-s10-t30-v5_4	5.177	325.571
han2-s10-t30-v5_3	4.278	266.207
han2-s10-t30-v5_2	4.849	299.664
han2-s10-t30-v5_1	4.661	306.017
han2-s10-t30-v5_0	5.117	327.222

TABLE II  
HYBRID AUTOMATA: A=2, S=10, T=10,30

Benchmark	BDD+SMT	AllMathSAT
han3-s10-t10-v5_9	4.620	T.O.
han3-s10-t10-v5_8	2.504	T.O.
han3-s10-t10-v5_7	3.681	T.O.
han3-s10-t10-v5_6	3.921	T.O.
han3-s10-t10-v5_5	2.771	T.O.
han3-s10-t10-v5_4	3.436	T.O.
han3-s10-t10-v5_3	3.139	T.O.
han3-s10-t10-v5_2	2.634	T.O.
han3-s10-t10-v5_1	4.073	T.O.
han3-s10-t10-v5_0	2.531	T.O.
han3-s10-t30-v5_9	222.872	T.O.
han3-s10-t30-v5_8	124.455	T.O.
han3-s10-t30-v5_7	126.316	T.O.
han3-s10-t30-v5_6	122.800	T.O.
han3-s10-t30-v5_5	109.060	T.O.
han3-s10-t30-v5_4	125.406	T.O.
han3-s10-t30-v5_3	175.055	T.O.
han3-s10-t30-v5_2	156.781	T.O.
han3-s10-t30-v5_1	146.164	T.O.
han3-s10-t30-v5_0	170.785	T.O.

TABLE IV  
HYBRID AUTOMATA: A = 3, S = 10, T = 10,30

500MB and a CPU time limit of 900sec. We also used a proper BDD variable ordering in the BDD+SMT experiments as to minimize the size of the BDDs for the boolean abstraction.

The results (in seconds) are reported in Tables VII-VI. Depending on the size of the problem, our approach is able to gain up to two orders of magnitude over cube enumeration.

We remark that our implementation is rather naive; in particular, we are not using any optimal ordering for BDD variable ordering, and the reported time includes the construction of the BDD for the boolean abstraction. Second, BDD-based enumeration does not exploit a wealth of preprocessing techniques that are in fact applied in AllMathSAT. Despite this, the experiments clearly shows the potential of the method.

## VIII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new approach to the computation of exact predicate abstractions. The approach

embeds reasoning with respect to the background theory within a BDD-based quantification algorithm, and is able to outperform approaches purely based on SMT solvers.

We plan to extend this work along different dimensions. First, we plan to investigate dedicated ordering heuristics, to take into account the individual variables occurring in the constrains, and to extend the algorithm to deal with conjunctively partitioned transition relations. Second, we will investigate the impact of incrementality of our approach in the setting of a CEGAR. Finally, we will apply the approach to the verification of timed and hybrid systems, as well as the verification of Verilog systems.

## ACKNOWLEDGEMENTS

We would like to thank Albert Oliveras for providing us with BarceLogic; thanks also to Aarti Gupta and to the other members of the MathSAT team for many useful discussions and insights.

Benchmark	BDD+SMT	AllMathSAT
han4-s5-t10-v5_9	.856	T.O.
han4-s5-t10-v5_8	1.183	T.O.
han4-s5-t10-v5_7	.678	T.O.
han4-s5-t10-v5_6	2.142	T.O.
han4-s5-t10-v5_5	.280	T.O.
han4-s5-t10-v5_4	1.576	T.O.
han4-s5-t10-v5_3	.441	T.O.
han4-s5-t10-v5_2	2.796	T.O.
han4-s5-t10-v5_1	.265	T.O.
han4-s5-t10-v5_0	.321	T.O.
han4-s5-t30-v5_9	705.020	T.O.
han4-s5-t30-v5_8	1.586	T.O.
han4-s5-t30-v5_7	11.112	T.O.
han4-s5-t30-v5_6	23.025	T.O.
han4-s5-t30-v5_5	2.808	T.O.
han4-s5-t30-v5_4	12.246	T.O.
han4-s5-t30-v5_3	5.597	T.O.
han4-s5-t30-v5_2	3.035	T.O.
han4-s5-t30-v5_1	188.458	T.O.
han4-s5-t30-v5_0	4.885	T.O.

TABLE V  
HYBRID AUTOMATA: A=4, S=5, T=10,30

Benchmark	BDD+SMT	AllMathSAT
han4-s10-t10-v5_9	6.359	T.O.
han4-s10-t10-v5_8	6.714	T.O.
han4-s10-t10-v5_7	5.527	T.O.
han4-s10-t10-v5_6	1.273	T.O.
han4-s10-t10-v5_5	7.498	T.O.
han4-s10-t10-v5_4	3.794	T.O.
han4-s10-t10-v5_3	2.787	T.O.
han4-s10-t10-v5_2	3.762	T.O.
han4-s10-t10-v5_1	5.235	T.O.
han4-s10-t10-v5_0	10.924	T.O.
han4-s10-t30-v5_9	486.974	T.O.
han4-s10-t30-v5_8	695.444	T.O.
han4-s10-t30-v5_7	594.663	T.O.
han4-s10-t30-v5_6	14.105	T.O.
han4-s10-t30-v5_5	331.554	T.O.
han4-s10-t30-v5_4	508.539	T.O.
han4-s10-t30-v5_3	371.207	T.O.
han4-s10-t30-v5_2	6.055	T.O.
han4-s10-t30-v5_1	234.709	T.O.
han4-s10-t30-v5_0	244.936	T.O.

TABLE VI  
HYBRID AUTOMATA: A=4, S=10, T=10,30

## REFERENCES

- [1] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample Guided Abstraction Refinement," in *CAV*, ser. LNCS, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.
- [2] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft," in *IFM*, ser. LNCS, E. A. Boiten, J. Derrick, and G. Smith, Eds., vol. 2999. Springer, 2004, pp. 1–20.
- [3] B. Cook, D. Kroening, and N. Sharygina, "Cogent: Accurate Theorem Proving for Program Verification," in *CAV*, 2005, pp. 296–300.
- [4] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate Abstraction of ANSI-C Programs Using SAT," *Formal Methods in System Design*, vol. 25, no. 2-3, pp. 105–127, 2004.
- [5] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*. LNCS 2648, Springer, 2003, pp. 235–239.
- [6] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog," in *Design Automation Conference (DAC)*, June 2005.
- [7] E. M. Clarke, H. Jain, and D. Kroening, "Verification of SpecC using Predicate Abstraction," in *MEMOCODE*, July 2004.
- [8] D. Kroening and N. Sharygina, "Image Computation and Predicate Refinement for RTL Verilog using Word Level Proofs," in *Proceedings of DATE 2007*, 2007, pp. 1325–1330.
- [9] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT Techniques for Fast Predicate Abstraction," in *CAV*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 424–437.
- [10] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.
- [11] A. Cimatti and R. Sebastiani, "Building Efficient Decision Procedures on top of SAT Solvers," in *Formal Methods for Hardware Verification*, ser. LNCS. Springer, 2006, no. 3965.
- [12] S. Kleene, *Mathematical Logic*. John Wiley and Sons Inc., 1967.
- [13] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM Press, 2001, pp. 530–535.
- [14] F. Somenzi, "CuDD v. 2.4.1," ftp://vlsi.colorado.edu/pub/cudd-2.4.1.tar.gz.
- [15] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," in *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, ser. LNCS, O. Grumberg, Ed., vol. 1254. Springer, 1997, pp. 72–83.
- [16] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani, "Efficient Theory Combination via Boolean Search," *Inf. Comput.*, vol. 204, no. 10, pp. 1493–1525, 2006.
- [17] D. Geist and I. Beer, "Efficient Model Checking by Automated Ordering of Transition Relation Partitions," in *Proceedings of Computer Aided Verification (CAV'94)*, ser. LNCS, D. L. Dill, Ed., no. 818. Stanford, California, USA: Springer, June 1994.
- [18] K. L. McMillan, "Applying SAT Methods in Unbounded Symbolic Model Checkin," in *CAV*, ser. LNCS, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 250–264.
- [19] B. Li, C. Wang, and F. Somenzi, "Abstraction Refinement in Symbolic Model Checking Using Satisfiability as the Only Decision Procedure," *STTT*, vol. 7, no. 2, pp. 143–155, 2005.
- [20] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring," in *ICCAD*. IEEE Computer Society / ACM, 2004, pp. 510–517.
- [21] O. Grumberg, A. Schuster, and A. Yagdar, "Hybrid BDD and All-SAT Method for Model Checking," in *In proceedings of Symposium on Satisfiability Solvers and Program Verification (SSPV)*, Seattle, USA, Aug. 2006, held in conjunction with FLOC'2006.
- [22] S. K. Lahiri, R. E. Bryant, and B. Cook, "A Symbolic Approach to Predicate Abstraction," in *CAV*, ser. LNCS, W. A. H. Jr. and F. Somenzi, Eds., vol. 2725. Springer, 2003, pp. 141–153.
- [23] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang, "Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement," in *CAV*, ser. LNCS, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 457–461.
- [24] S. K. Lahiri, T. Ball, and B. Cook, "Predicate abstraction via symbolic decision procedures," in *CAV*, ser. LNCS, A. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 24–38.
- [25] D. Déharbe and S. Ranise, "Light-Weight Theorem Proving for Debugging and Verifying Units of Code," in *SEFM*. IEEE Computer Society, 2003, pp. 220–228.
- [26] A. Armando, "Simplifying OBDDs in Decidable Theories," in *In Proceedings of Pragmatics of Decision Procedures in Automated Reasoning 2003*, Miami, USA, Jul. 2003, A Workshop Affiliated to CADE 19.
- [27] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard, "Fully Symbolic Model Checking of Timed Systems Using Difference Decision Diagrams," in *Workshop on Symbolic Model Checking*, vol. 23, no. 2, The IT University of Copenhagen, Denmark, Jun. 1999.
- [28] M. P. Schuijers, "Integrating a BDD Prover and a DPLL SAT Solver for Abstract Data Types," Technische Universiteit Eindhoven, Technical Report, Jan. 2006.
- [29] A. Cimatti, M. Roveri, V. Schuppan, and S. Tonetta, "Boolean Abstraction for Temporal Logic Satisfiability," in *Proceedings of CAV'07*. Berlin, Germany: Springer, Jul. 2007.

# Induction in CEGAR for Detecting Counterexamples

Chao Wang, Aarti Gupta, and Franjo Ivančić  
 NEC Labs America, Princeton, NJ 08540, U.S.A.

{chaowang, agupta, ivancic}@nec-labs.com

**Abstract**—Induction has been studied in model checking for proving the validity of safety properties, i.e., showing the *absence* of counterexamples. To our knowledge, induction has not been used to refute safety properties. Existing algorithms including bounded model checking, predicate abstraction, and interpolation are not efficient in detecting long counterexamples. In this paper, we propose the use of induction inside the counterexample guided abstraction and refinement (CEGAR) loop to prove the *existence* of counterexamples. We target bugs whose counterexamples are long and yet can be captured by regular patterns. We identify the pattern algorithmically by analyzing the sequence of spurious counterexamples generated in the CEGAR loop, and perform the induction proof automatically. The new method has little additional overhead to CEGAR and this overhead is insensitive to the actual length of the concrete counterexample.

## I. INTRODUCTION

Induction techniques have been used in model checking to prove safety properties in a state transition system. A property is called *safety* if it can be refuted by examining a finite computation path of the model. Properties of the form  $AG \psi$  (i.e.,  $\psi$  is an invariant) are an important special case since a general safety property can be reduced to an invariant by a compilation process [1]. Conceptually, one can prove invariant properties by showing that  $\psi$  holds in the initial states, and  $\psi$  is maintained by the transition relation. SAT based induction methods [2], [3], for instance, rely on the observation that a failing property has a simple path from an initial state to a bad state. An invariant holds if all paths of length  $k$  or shorter satisfy  $\psi$ , and there is no simple path of length  $k + 1$  or longer from an initial state. Induction has a clear advantage over other proof methods since it has to consider only paths of a shorter length (up to  $k$  in  $k$ -induction) whereas bounded model checking (BMC [4]), for instance, needs to check all paths up to a completeness threshold.

However, induction has not been used to refute safety properties, that is, to show that a concrete counterexample exists. Existing methods for finding bugs, including BMC, counterexample guided abstraction refinement (CEGAR [5], [6], [7]), and interpolation [8], are not efficient in the presence of long concrete counterexamples. For example, BMC has been widely regarded as effective in detecting shallow bugs in large models; however, state-of-the-art BMC algorithms handle only up to a few hundred unrollings on typical industrial-scale models. When there are deep bugs, CEGAR and interpolation based methods do not work well either, since they too rely on finding a state-by-state match between the abstract and the concrete counterexamples.

We propose an induction based refutation method for detecting long counterexamples, whose computational overhead

```
(1) unsigned i , n ;           (1) bool P = * ;
(2) ...                       (2) ...
(3) n = 10000 ;              (3) P = * ;
(4) ...                       (4) ...
(5) i = 0 ;                  (5) P = T ;
(6) while ( i<=n ) {         (6) while ( * ) {
(7)     assert ( i<n ) ;     (7)     assert ( P ) ;
(8)     i++ ;                (8)     P = P ? * : F ;
}                             }
```

Fig. 1. The original C program and the first Boolean abstraction

is independent of the actual counterexample length. Our main observation is that deep bugs can often be captured by a regular pattern in the counterexample of a family of systems obtained by introducing a parameter to the system under analysis. Instead of looking for a state-by-state match of the abstract counterexample to a particular concrete counterexample, we prove by induction that there always exists a counterexample of that general pattern.

Consider the program in Fig. 1, which has a simple and yet representative bug in line 7 (e.g., an array bound violation). Detecting this bug is challenging for all aforementioned methods. For illustration purposes, a standard predicate abstraction procedure would add the following predicates,  $P: (i < 10000)$ ,  $P1: (i < 9999)$ ,  $P2: (i < 9998)$ , etc. The procedure needs  $n$  refinement iterations in order to produce a concrete counterexample. One may argue that CEGAR is not well suited for finding this type of bugs. However, if the loop condition were  $(i < n)$  and the assertion never failed, then CEGAR is efficient in getting the proof.

Our new method provides complementary strength (good for refutation) to the popular CEGAR style abstraction algorithms (good for proof). In Fig. 1, regardless of the initial value of  $n$ , there is an assertion failure in Line 7. Furthermore, the sequence of concrete counterexamples, in terms of the line numbers leading to the failure, is

$$(1) (2) (3) (4) (5) \quad \{ (6) (7) (8) \}^n \quad (6) (7)$$

If we can prove that a counterexample of this regular pattern exists for all  $n \geq 1$ , then it follows that there exists a counterexample for  $n = 10000$ .

Although CEGAR is not efficient in detecting long counterexamples, it can be useful in identifying the regular pattern of the counterexample. We present an algorithm to identify the regular pattern of the counterexample, by analyzing the failed counterexample concretization attempts inside the CEGAR loop. The basic idea behind this algorithm is that the regular pattern of a concrete counterexample is often shown in the series of spurious counterexamples encountered in the CEGAR loop. In Fig. 1, for instance, the set of spurious abstract counterexamples produced by the CEGAR procedure

have the same regular pattern—they differ from the concrete counterexample only in the number of copies of the recurring segment (6) (7) (8).

The existence of a parameterized counterexample can then be proved by induction: (1) in the base step, we show that a concrete counterexample of the given regular pattern exists for  $n = 1$ ; (2) as an induction hypothesis, we assume that for  $n = k$ , a concrete counterexample of the given pattern exists. (3) in the induction step, we extend the counterexample for  $n = k$  to build a new counterexample for  $n = k + 1$ , and show that the new counterexample exists. One of our main contributions is proposing a *goal containment* check that is sufficient to establish the induction proof for  $n = k + 1$  based on the induction hypothesis for  $n = k$ . In the goal containment check, we align the common prefixes of the two consecutive counterexamples for  $k$  and  $k + 1$  and compare their suffixes.

We have implemented the proposed method in a standard CEGAR procedure and conducted experiments on some software examples from the public domain. The results show that when augmented with the new induction based method, CEGAR is able to find some very long counterexamples in nontrivial examples, most of which would have eluded existing model checking methods.

The rest of this paper is organized as follows. After reviewing the related work and introducing the notation, we present the algorithm for identifying a counterexample pattern in Section IV. We present our symbolic method for establishing an induction proof in Section V. We demonstrate the effectiveness of our method through experiments in Section VII, and then conclude this paper in Section VIII.

## II. RELATED WORK

Detecting a long counterexample has been a well-known problem in formal verification of both hardware and software systems. In [9], Ho *et al.* attempt to solve the problem by simulating up to a deep state and then searching around that state exhaustively. This technique can be considered as a semi-formal method, in that it combines directed random simulation and model checking. A similar approach was adopted by DART [10] in the context of program verification. Semi-formal methods may miss bugs since they selectively, as opposed to exhaustively, explore the state space.

In [11], Nanshi and Somenzi use guided pseudo-random simulation in the search of a concrete counterexample, where the guidance is provided by synchronous onion rings (SORs [12]), a data structure that implicitly captures all shortest abstract counterexamples. In [13], Bjesse and Kukula present a repeat extender algorithm for counterexample generation within an abstraction refinement loop. They use the abstract counterexample as *backshell lighthouses* without limiting the BMC search to concrete counterexamples of the same length. In [14], Kroening and Weissenbacher propose a similar method targeting counterexamples with loops. After heuristically identifying the loop, they put the assignment statements of the loop body into a closed form representation, and use a SAT solver to calculate a conservative bound on the number of loop iterations. Then, they use BMC to iterate

through the loop exactly that number of times and derive a concrete counterexample.

All these CEGAR based methods insist on finding a concrete path from an initial state leading to the bad state, which makes them less scalable when the model is complex and the counterexample is long.

In [15], Seghir and Podelski use *transition abstraction* to shortcut the “transfinite” sequence of refinement steps. They abstract not just states but also the state changes induced by the structured language constructs including `for` and `while` statements. In [16], Ball *et al.* analyze termination of loops without refinement and without well-founded sets and ranking functions. Their method is based on the symbolic reasoning of abstract counterexamples; instead of using induction, they try to identify *must transitions* in the abstract state transition graph using conditions on the structure of the graph.

## III. PRELIMINARIES

### A. From CDFGs to Models

We represent the model under verification as a tuple  $M = \langle S, T, I, S_E \rangle$ , where  $S$  is a set of states,  $T \subseteq S \times S$  is the transition relation,  $I \subseteq S$  is the set of initial states, and  $S_E \subseteq S$  is the set of error states. Given a set  $X = \{x_1, \dots, x_n\}$  of state variables, each state  $s \in S$  is a valuation of the variables in  $X$ . A concrete path is a sequence of states  $s_1 \dots s_l$  such that  $(s_i, s_{i+1}) \in T$  for all  $1 \leq i < l$ .

We use a control and data flow graph (CDFG) as the intermediate representation, where CDFGs may be derived from either hardware designs or software programs [17].

**Definition 1** A control and data flow graph (CDFG) is a 5-tuple  $G = \langle \mathcal{B}, \mathcal{E}, V, \Delta, \theta \rangle$  such that

- $\mathcal{B} = \{b_1, \dots, b_L\}$  is a finite set of basic blocks, where  $b_1$  is the entry block.
- $\mathcal{E} \subseteq \mathcal{B} \times \mathcal{B}$  is a set of edges representing transitions between basic blocks.
- $V$  is a finite set of variables that consists of actual design/program variables and the auxiliary variables added for modeling the hardware/software semantics.
- $\Delta : \mathcal{B} \rightarrow 2^{S_{asgn}}$  is a labeling function that labels each basic block with a set of parallel assignments.  $S_{asgn}$  is the set of possible assignments.
- $\theta : \mathcal{E} \rightarrow S_{cond}$  is a labeling function that labels each edge with a conditional expression.  $S_{cond}$  is the set of possible conditional expressions.

Figure 2 shows a sample C program and its CDFG. Each rectangle is a basic block. Block 1 is the entry block and block 7 is the error block. Basic blocks are connected with each other by edges, which are labeled by conditional expressions. For example, the transition from block 3 to block 4 is guarded by  $(a < 100)$ . Edges that are not labeled by any condition are assumed to have a true label.

The CDFG is regarded as an explicit representation of the concrete model. To represent the CDFG symbolically as a verification model  $M = \langle S, T, I, S_E \rangle$ , we add a special Program Counter (PC) variable  $x_{pc}$  whose domain is the set

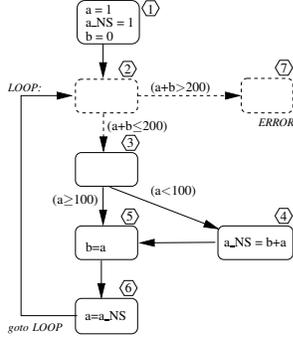


Fig. 2. The control and data flow graph

```

void main ()
{
  a = 1;
  a_#NS = 1;
  b = 0;

  LOOP:
  if (a + b > 200)
    ERROR

  if (a < 100)
    a_#NS = b + a;
    b = a;

  a = a_#NS;
  goto LOOP;
}

```

of basic block indexes  $\mathcal{B} = \{b_1, \dots, b_n\}$ . Furthermore, we assume in the model  $M$  that  $X = \{x_{pc}\} \cup V$  is the set of state variables ( $V$  consists of the original variables from the model). If the entry block is  $b_1 \in \mathcal{B}$  and the error block is  $b_{err} \in \mathcal{B}$ , then we have  $I = (x_{pc} = b_1)$  and  $S_E = (x_{pc} = b_{err})$ .

### B. Parameterized Counterexamples

An abstract model  $\widehat{M}$  can be derived from an over-approximated pair  $(\widehat{T}, \widehat{I})$  such that  $T \subseteq \widehat{T}$  and  $I \subseteq \widehat{I}$ . Given a CDFG representation, a natural way of creating an abstract model  $\widehat{M}$  is to over-approximate the assignment statements. For instance, if we ignore the assignment statements (by assuming that arbitrary values can be assigned to the left-hand side variables), the remaining control flow graph can be regarded as an abstract model. In this abstract model  $\widehat{M}$ , an abstract state is represented by the set  $\widehat{s}_i = (x_{pc} = b_i)$ , where  $b_i \in \mathcal{B}$ . Since all the guards in  $\widehat{M}$  can be either true or false,  $\widehat{M}$  has all the behaviors of  $M$  and possibly more. An abstract path is a sequence of abstract states  $\pi = \widehat{s}_1, \dots, \widehat{s}_j$ .

**Definition 2** A parameterized abstract counterexample  $\xi$  is an abstract path of the form

$$\xi = \widehat{s}_1 \dots \widehat{s}_{i-1} \{ \widehat{s}_i \dots \widehat{s}_j \}^n \widehat{s}_{j+1} \dots \widehat{s}_l,$$

such that  $\widehat{s}_1 \subseteq \widehat{I}$  and  $\widehat{s}_l \subseteq \widehat{S}_E$ . The recurring segment is  $\pi_r = \widehat{s}_i \dots \widehat{s}_j$ , the finite prefix is  $\pi_p = \widehat{s}_1 \dots \widehat{s}_{i-1}$ , and the finite suffix is  $\pi_s = \widehat{s}_{j+1} \dots \widehat{s}_l$ . The integer  $n$  is the number of copies of  $\pi_r$ .

Any of the three segments in  $\xi$  can be empty. For example, if  $\xi$  is used to match the concrete counterexample in a state-by-state way, we assume that  $\pi_r$  may be an empty segment.

Given a set  $Q$  of states, the post-condition (pre-condition) of  $Q$  with respect to  $T$  consists of all successors (predecessors) of  $Q$ . Formally,

$$\begin{aligned} pre(Q) &= \{s \mid \exists s' \in Q : (s, s') \in T\}, \\ post(Q) &= \{s' \mid \exists s \in Q : (s, s') \in T\}. \end{aligned}$$

The definition can be extended transitively with respect to a control path  $\pi = \widehat{s}_1, \dots, \widehat{s}_j$  as follows,

$$\begin{aligned} pre^*(\pi, Q) &= \{s \mid \text{there is a concrete run inside } \pi \\ &\quad \text{from } s \text{ to } s' \text{ such that } s' \in (Q \cap \widehat{s}_j)\}, \\ post^*(\pi, Q) &= \{s' \mid \text{there is a concrete run inside } \pi \\ &\quad \text{from } s \text{ to } s' \text{ such that } s \in (Q \cap \widehat{s}_1)\}. \end{aligned}$$

Given a control path  $\pi$ , the transitive version of pre-condition can be computed as follows: let  $Z = \widehat{s}_j \cap Q$ , and then repeatedly compute  $Z = \widehat{s}_i \cap pre(Z)$  for all  $i \leq j$ .

Function  $f(V)$  is called *disjunctively decomposable* with respect to the partition  $V = V_a \cup V_b$  if

$$f(V) = (\exists V_a. f(V_a, V_b)) \wedge (\exists V_b. f(V_a, V_b)),$$

### C. The CEGAR Procedure

Counterexample guided abstraction refinement is an iterative procedure consisting of three phases: abstraction, model checking, and refinement. Algorithm 1 shows the pseudo code of a generic CEGAR procedure. Typically, one starts with a coarse initial abstraction  $\widehat{M}$  and applies model checking. If the property holds in  $\widehat{M}$ , it also holds in  $M$  and the property is proved. If the property fails and the model checker returns an abstract counterexample (ACE), a concretization procedure is used to check whether a concrete counterexample (CCE) exists. If a concrete counterexample exists, the property is refuted. Otherwise, the *spurious* counterexample is used during refinement to identify the needed information currently missing in the abstraction.

#### Algorithm 1 CEGAR( $M, \psi$ )

```

1:  $\widehat{M} = \text{INITIAL\_ABSTRACTION}(M, \psi)$ ;
2: while (1) do
3:    $ACE = \text{MODEL\_CHECKING}(\widehat{M}, \psi)$ ;
4:   if ( $ACE$  is empty) then
5:     return TRUE;
6:   end if
7:    $CCE = \text{CONCRETIZECEX}(M, ACE)$ ;
8:   if ( $CCE$  not empty) then
9:     return (FALSE,  $CCE$ );
10:  end if
11:   $\widehat{M} = \text{REFINEMENT}(M, ACE)$ ;
12: end while

```

One inefficiency of CEGAR in detecting long counterexamples is due to its concretization algorithm. Algorithm 2 is a standard concretization procedure which takes the abstract counterexample  $\pi = \widehat{s}_1, \dots, \widehat{s}_l$  as input. It starts from the error states  $\widehat{s}_l$  and repeatedly computes pre-conditions. If the pre-condition  $q_1$  is non-empty, a concrete counterexample has been found. If the precondition becomes empty before  $i$  decreases to 1, concretization fails and the abstract counterexample is marked as *spurious*. After the counterexample is declared as spurious, the refinement step follows to improve the abstract model by analyzing the spurious counterexample. The CEGAR loop continues until either the property is proved (no abstract counterexample), or a concrete counterexample is found, or the procedure runs out of computing resources.

It may take a large number of refinement iterations for the CEGAR procedure to produce an abstract counterexample whose length matches the length of the concrete counterexample—before that, all the abstract counterexamples are declared as spurious. If the concrete counterexample has a parameterized pattern, using Algorithm 2 to concretize it is inefficient. Recall that  $\xi = \pi_p \{ \pi_r \}^n \pi_s$ . When  $\xi$  contains a large number of copies of  $\pi_r$ , the pre- and post-condition computations over  $\xi$  can be expensive.

---

**Algorithm 2** CONCRETIZECEX( $M, \pi$ )

---

```
1:  $i = l$ ;  
2:  $q_i = \widehat{s}_i$ ;  
3: while ( $i \neq 1$ ) do  
4:    $q_{i-1} = \text{pre}(q_i) \cap \widehat{s}_{i-1}$ ;  
5:   if ( $q_{i-1} \neq \emptyset$ ) then  
6:      $i = i - 1$ ;  
7:   else  
8:     return no concrete counterexample;  
9:   end if  
10: end while  
11: return a concrete counterexample;
```

---

## IV. IDENTIFYING COUNTEREXAMPLE PATTERNS

To augment the standard CEGAR procedure, we add an induction proof attempt right after concretization fails, but before refinement. Our procedure tries to identify a regular pattern from the set of abstract counterexamples by analyzing the failed concretization attempts. The counterexample pattern becomes a hypothesis, which subsequently will be validated by an induction step. If we can prove that for all induction parameter values (including the one in the concrete counterexample), an instance of the parameterized counterexample exists, then the property is refuted. If this added induction proof attempt does not succeed, we fall back upon the standard CEGAR loop and continue with refinement.

## A. The Recurring Segment

To characterize  $\xi$ , we need to identify the head and tail states of  $\pi_r$  from a given abstract path  $\pi = \widehat{s}_1 \dots \widehat{s}_l$ . We accomplish this by modifying the standard concretization procedure. We rely on the fact that if the concrete counterexample is an instance of a parameterized abstract counterexample  $\pi_p \{ \pi_r \}^n \pi_s$ , then the CEGAR procedure is likely to generate a series of spurious counterexamples of the following form:

```
iteration 1:  $\pi = \pi_p \pi_r \pi_s$ ,  
iteration 2:  $\pi = \pi_p \pi_r \pi_r \pi_s$ ,  
iteration 3:  $\pi = \pi_p \pi_r \pi_r \pi_r \pi_s$ ,  
...
```

the sequence continues until the number of copies of  $\pi_r$  matches the value in the concrete counterexample.

Recall that for a spurious abstract path  $\pi = \widehat{s}_1 \dots \widehat{s}_l$ , the concretization procedure in Algorithm 2 will find a *failing index*  $i$  such that  $1 \leq i < l$  and  $\text{pre}(q_{i+1}) \cap \widehat{s}_i = \emptyset$ . In Fig. 3, for instance, the failing index is  $i$  since  $q_i$  is empty. In Algorithm 2, once a failing index  $i$  is found,  $\pi$  is declared as spurious and the concretization stops.

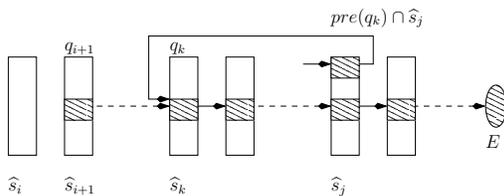


Fig. 3. Backleap to identify the counterexample pattern

We modify Algorithm 2 to allow the search for a potentially longer concrete counterexample by using a “backleap” strategy. The new concretization procedure is given in Algorithm 3, in which the additional steps (with respect to the standard algorithm) are listed in lines 8-19.  $C_{thres}$  is a predetermined threshold denoting the maximum number of backleaps allowed in a concretization attempt.

The idea is to start from the failing index  $i$  and search backward for a transition  $(\widehat{s}_j, \widehat{s}_k)$  with  $i \leq k \leq j \leq l$  such that in the concrete model  $\widehat{s}_j$  is reachable from  $\widehat{s}_k$  in one step. If  $k$  and  $j$  exist, the concretization retreats from  $q_{i+1}$  back to  $q_k$  and makes a successful backleap from  $q_k$  to  $\widehat{s}_j$ ; after that we continue the concretization process from  $\text{pre}(q_k) \cap \widehat{s}_j$ . At the same time, we record  $\widehat{s}_j$  as a candidate tail state of  $\pi_r$  and  $\widehat{s}_k$  as a candidate head state of  $\pi_r$ . In this modified concretization procedure, we can make backleaps more than once (bounded by the constant  $C_{thres}$ )—therefore, it is possible to find a concrete counterexample that is longer than the abstract counterexample to be concretized.

---

**Algorithm 3** CONCRETIZECEX\_BACKLEAP( $M, \pi$ )

---

```
1:  $i = l$ ;  $q_i = \widehat{s}_i$ ;  
2:  $\eta_{bLeap} = 0$ ;  
3: while ( $i \neq 1$ ) do  
4:    $q_{i-1} = \text{pre}(q_i) \cap \widehat{s}_{i-1}$ ;  
5:   if ( $q_{i-1} \neq \emptyset$ ) then  
6:      $i = i - 1$ ;  
7:   else  
8:     find  $k$  and  $j$  such that  $i \leq k \leq j \leq l$  and  $\text{pre}(q_k) \cap \widehat{s}_j \neq \emptyset$ .  
9:     if ( $k$  and  $j$  do not exist, or  $\eta_{bLeap} > C_{thres}$ ) then  
10:      if ( $\text{PROVE\_CEX\_BY\_INDUCTION}()$ ) then  
11:        return a concrete counterexample; //proved  
12:      else  
13:        return no concrete counterexample;  
14:      end if  
15:    end if  
16:     $q_j = \text{pre}(q_k) \cap \widehat{s}_j$ ;  
17:     $i = j$ ;  
18:     $\eta_{bLeap} + +$ ;  
19:    add segment  $\pi^{k,j}$  to a list of candidates of  $\pi_r$ ;  
20:  end if  
21: end while  
22: return a concrete counterexample;
```

---

If after making  $C_{thres}$  backleaps, the new concretization procedure fails to find a concrete counterexample but some  $\pi_r = \widehat{s}_k \dots \widehat{s}_j$  have been recognized, we enter the induction proof mode (lines 9-15). In Algorithm 3, induction proof is implemented in the function  $\text{PROVE\_CEX\_BY\_INDUCTION}$ .

## B. The Induction Parameter

Given a recurring segment  $\pi_r = \widehat{s}_k \dots \widehat{s}_j$ , we need to identify the *potential* induction parameter associated with  $\pi_r$  before calling  $\text{PROVE\_CEX\_BY\_INDUCTION}$ . We will show in the next section that checking whether the induction proof holds is cheap computationally (compared to model checking), and only a correct induction parameter (paired with a true recurring segment) allows the induction proof to hold. The naive approach is to blindly try all the program variables one by one as the induction parameter, and under that assumption check whether the induction proof holds. The naive approach can be costly, but it does not affect the correctness of the overall CEGAR procedure.

In practice, however, we need to reduce the overhead of identifying induction variable. We first compute a list of promising candidate variables, that is, the program variables in  $V$  that are likely to be induction parameters. Then we try the candidate variables one by one, to see whether treating each of them as the induction parameter makes the induction proof go through.

We use the following criteria to filter out non-induction variables. Let  $\pi_r = \widehat{s}_k, \dots, \widehat{s}_j$  and  $g(V)$  be the conditional expression guarding the transition from  $\widehat{s}_j$  to  $\widehat{s}_k$  (the back edge). For a program variable  $n$  to be the induction parameter,  $n$  needs to be in the transitive support of the expression  $g(V)$ . Furthermore, all the guard expressions inside  $\pi_r$  must be monotonic with respect to variable  $n$ —this guarantees that as long as a transition (of  $\pi_r$ ) is valid for  $n = k$ , it is valid also for the  $n = k + 1$  counterexample instance.

### C. Counterexample Instances

In order to prove that  $\xi$  exists for all  $n \geq 1$ , we need to control the value of  $n$  in the model to produce a set of parameterized counterexample instances. Conceptually this is accomplished by finding inside  $\pi_p$  the last assignment statement to  $n$ , and replacing it with a statement assigning a symbolic value  $k$  to  $n$ .

In Fig. 1, for instance, the prefix segment is  $\pi_p = (1) (2) (3) (4) (5)$  and the last assignment statement to  $n$  is in line 3. A simple static analysis can locate the statement  $n=1000$  at line 3 and rewrite it into  $n=k$ , where  $k$  remains a parameterized symbolic value. When setting  $k$  to 1, we can check whether  $\xi$  exists for the induction basis  $n=1$ .

In practice, when  $\xi$  and  $n$  are given, we have implemented a procedure to locate the last assignment statement to  $n$  inside  $\pi_p$ , followed by an automatic rewriting of CDFG representation of the concrete model<sup>1</sup>.

## V. PROVING THE EXISTENCE OF COUNTEREXAMPLES

In this section we explain the underlying algorithm for the function `PROVE_CEX_BY_INDUCTION`. Recall that before entering the induction proof mode, we have already identified a potential parameterized abstract counterexample  $\xi = \pi_p \{ \pi_r \}^n \pi_s$ .

### A. Induction Proof

The induction hypothesis is that there exists a concrete path inside the abstract counterexample  $\pi_p \{ \pi_r \}^k \pi_s$ . We divide the counterexample instance into  $\pi_p \{ \pi_r \}^k$  and  $\pi_s$ , and define the intermediate pre-condition and post-conditions as follows:

$$\begin{aligned} G &= \text{post}^*(\pi_p, I) \\ F &= \text{post}^*(\pi_p \{ \pi_r \}^k, I) \\ B &= \text{pre}^*(\pi_s, S_E) \end{aligned}$$

To rephrase the induction hypothesis,

$$F \cap B \neq \emptyset .$$

<sup>1</sup>If  $n$  does not appear in  $\pi_p$ , e.g., when loop condition is  $(i < \text{CONST})$  and we rewrite it as  $(i < n)$ , we assume that  $n = k$  holds in the initial state.

That is, there exists  $s \in (F \cap B)$  such that  $s$  is reachable from  $I$  through  $\pi_p \{ \pi_r \}^k$  and can reach  $S_E$  through  $\pi_s$ .

We want to prove that there also exists a concrete path inside the abstract counterexample  $\pi_p \{ \pi_r \}^{k+1} \pi_s$ . Similar to the previous case, we divide the counterexample instance into two parts:  $\pi_p \{ \pi_r \}^k$  and  $\pi_r \pi_s$ . The corresponding pre-conditions and post-conditions are defined as follows:

$$\begin{aligned} G' &= \text{post}^*(\pi_p, I) \\ F' &= \text{post}^*(\pi_p \{ \pi_r \}^k, I) \\ B' &= \text{pre}^*(\pi_r \pi_s, S_E) \end{aligned}$$

The existence of a concrete counterexample is rephrased as follows,

$$F' \cap B' \neq \emptyset .$$

The induction step says that, if there exists a concrete state  $s \in (F \cap B)$  when  $n = k$ , then there exists a concrete state  $s' \in (F' \cap B')$  when  $n = k + 1$ . With a little abuse of notation ( $F$  as a set and  $F(V)$  as a formula), we express the induction step formally as follows,

$$\exists V . F(V) \wedge B(V) \rightarrow \exists V . F'(V) \wedge B'(V) .$$

Here  $V$  is the set of program variables. However, we shall never compute  $F'(V)$  and  $F(V)$  explicitly since they are over parameterized segment (expensive to compute). Instead, we rely on the analysis of  $G(V)$  and  $G'(V)$  to derive sufficient conditions under which the induction holds.

### B. Induction Condition

We partition the set  $V$  of variables into  $V = V_a \cup V_b$ .  $V_b$  contains the induction parameter  $n$  and variables which are assigned in  $\pi_p$  to values that depends on  $n$ , and  $V_a$  contains the remaining variables. To get an induction proof, consider the following requirement on  $V_a$  and  $V_b$  (**C0**):

- in  $\pi_r$ , variables in  $V_b$  do not appear in any assignment (neither in left-hand side nor in right-hand side);
- guards  $g(V_a, V_b)$  in  $\pi_r$  are monotonic with respect to  $V_b$ .

In such cases, pre- and post-conditions over  $\pi_r$  can be computed by updating functions for  $V_a$  and  $V_b$  separately (a Cartesian product). This characteristic has been captured by the notion of a disjunctively decomposable function described in Section III.

Given  $G$  and  $G'$ , we define

$$\begin{aligned} G_a &= \exists V_b . G(V_a, V_b) \\ G_b &= \exists V_a . G(V_a, V_b) \\ G'_a &= \exists V_b . G'(V_a, V_b) \\ G'_b &= \exists V_a . G'(V_a, V_b) \end{aligned}$$

The induction holds if the following conditions are satisfied:

**C1:**  $G$  and  $G'$  differ only in the valuations of  $V_b$ :

$$\begin{aligned} G &= G_a(V_a) \wedge G_b(V_b), \\ G' &= G'_a(V_a) \wedge G'_b(V_b), \\ G_a &= G'_a \end{aligned}$$

**C2:**  $B$  and  $B'$  satisfy the following *goal containment* check:

$$\exists V_b . G_b \wedge B \rightarrow \exists V_b . G'_b \wedge B' .$$

All conditions can be checked algorithmically in the CDFG model by a combination of static analysis (for checking the partition of  $V$ ) and pre-condition and post-condition computations over finite counterexample segments.

### C. Proof of Correctness

The correctness of the induction proof is established by Theorem 3. The proof is illustrated pictorially in Fig. 4.

**Theorem 3** *If  $F \cap B \neq \emptyset$  and conditions **C0,C1,C2** are satisfied, then  $F' \cap B' \neq \emptyset$ .*

*Proof:* Since  $G$  and  $G'$  are disjunctively decomposable,  $V_b$  variables do not change their values in  $\pi_r$ , and  $V_a$  variables are updated independently from  $V_b$  in  $\pi_r$ , we know that  $F$  and  $F'$  are also disjunctively decomposable; that is,

$$\begin{aligned} F &= F_a(V_a) \wedge F_b(V_b) = F_a(V_a) \wedge G_b(V_b), \\ F' &= F'_a(V_a) \wedge F'_b(V_b) = F'_a(V_a) \wedge G'_b(V_b), \end{aligned}$$

and  $F_a = F'_a$  (because of  $G_a = G'_a$ ). From condition **C2**,

$$\begin{aligned} \exists V_b. G_b \wedge B &\subseteq \exists V_b. G'_b \wedge B' , \\ \exists V_b. F_a(V_a) \wedge G_b \wedge B &\subseteq \exists V_b. F'_a(V_a) \wedge G'_b \wedge B' , \\ \exists V_b. F \wedge B &\subseteq \exists V_b. F' \wedge B' . \\ \exists V. F(V) \wedge B(V) &\subseteq \exists V. F'(V) \wedge B'(V) . \end{aligned}$$

This means that if  $s \in F \cap B$  exists, then  $s' \in F' \cap B'$  exists. Note that  $s$  and  $s'$  may differ only in their valuations of variables in  $V_b$ . ■

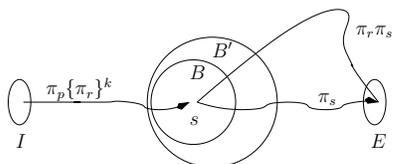


Fig. 4. The normal induction condition.

## VI. GOAL CONTAINMENT CHECKING

### A. The Working Example

In Fig. 1, when the set of predicates is empty (the initial abstraction), the abstract counterexample produced by the CEGAR procedure is  $\pi = (1) \dots (5) (6) (7)$ . This abstract counterexample cannot be concretized by the standard concretization procedure or our backleap algorithm. Furthermore, our concretization algorithm cannot detect any counterexample pattern; at line 8 in Algorithm 3 there does not exist a valid index pair  $(j, k)$  for backleap.

After the first refinement iteration (which removes the spurious counterexample), the induction proof attempt becomes possible. Algorithm 3 will return the following counterexample pattern:  $\xi = \pi_p \{ \pi_r \}^n \pi_s$  such that  $\pi_p = (1) \dots (5)$ ,  $\pi_r = (6) (7) (8)$ , and  $\pi_s = (6) (7)$ .

The set  $V$  of program variables is partitioned into  $V_a = \{i\}$  and  $V_b = \{n\}$ . Inside  $\pi_r$ , all the LHS variables are included in  $V_a$ , and the only variable in  $V_b$  does not change. Furthermore, inside  $\pi_r$  the guard  $g : (i \leq n)$  is monotonic with respect to  $n$ .

For the first two conditions,

$$\begin{aligned} G &= (i = 0) \wedge (n = k) , \\ G' &= (i = 0) \wedge (n = k + 1) . \end{aligned}$$

Let  $G_a = G'_a = (i = 0)$ ,  $G_b = (n = k)$ , and  $G'_b = (n = k + 1)$ ; both  $G$  and  $G'$  are disjunctively decomposable.

For the last condition,

$$\begin{aligned} B &= pre^*(\pi_s, i \geq n) = (i = n) \\ B' &= pre^*(\pi_r \pi_s, i \geq n) = (i + 1 = n) \end{aligned}$$

Therefore,

$$\begin{aligned} \exists V_b. G_b \wedge B &= (i = k) \\ \exists V_b. G'_b \wedge B' &= (i + 1 = k + 1) \end{aligned}$$

This proves the goal containment,

$$\exists V_b. G_b \wedge B \rightarrow \exists V_b. G'_b \wedge B' .$$

### B. Goal Containment

Sets  $B'$  and  $B$  are regarded as the *goals* of postcondition computations over the common prefix  $\pi_p \{ \pi_r \}^k$ . The goal containment check requires a decision procedure that supports quantified formulas. In our implementation, we use a mixed model checking procedure [18] which incorporates both bit-level and word-level symbolic representations. If all program variables are assumed to be in finite domains, one can also choose to use standard BDD-based fixpoint algorithms

Since we always consider a single program path (a finite prefix or suffix), the pre- and post-condition computations can be made efficient in practice. Although the CDFG may have many branching statements, when computing  $post^*(\cdot)$  and  $pre^*(\cdot)$  over  $\pi_p$  and  $\pi_s$ , we do not need to consider more than one branch at each *pre* or *post* step. For instance, given  $\pi = s_i, \dots, s_j$  and a propositional formula  $\phi$ , the weakest liberal pre-condition [19] of  $\phi$  with respect to  $\pi$ , denoted by  $wlp(\pi, \phi)$ , is computed as follows,

- For a statement  $s : v = e$ ,  $wlp(s, \phi) = \phi(e/v)$ ;
- For a statement  $s : \text{assume}(c)$ ,  $wlp(s, \phi) = \phi \wedge c$ ;
- For a sequence of statements  $s_1 ; s_2$ ,  $wlp(s_1 : s_2, \phi) = wlp(s_1, wlp(s_2, \phi))$ .

For a single CDFG path, there are only two types of statements: assignment statements and branching statements ( $\text{assume}(c)$  comes from  $\text{if}(c)$ ). Complex C statements involving pointers, arrays, structures, function calls, etc. can be rewritten into simple statements involving scalar variables only during a preprocessing phase of the CDFG representation [20]. Therefore, the pre- and post-condition results over a single CDFG path do not blow up. In practice, the time spent on computing  $G, G', B, B'$  is often negligible when compared to other phases of the CEGAR procedure.

### C. Strengthening Induction

The conditions can be strengthened by imposing a restricted area within which goal containment should hold. This step is optional, but may increase the chance of getting a proof. Specifically, we identify a state subspace  $F_\infty \subseteq S$  such that goal containment within  $F_\infty$  can establish the proof. We define  $F_\infty$  as the union of  $F$  for all  $k \geq 1$ ,

$$F_\infty = \bigcup_{k=1}^{\infty} \text{post}(\pi_p\{\pi_r\}^k, I) .$$

By definition,  $F \wedge F_\infty = F$  and  $F' \wedge F_\infty = F'$ .

Assume that  $F \wedge B \neq \emptyset$ . We now prove that if  $\exists V_b. (G_b \wedge B \wedge F_\infty) \rightarrow \exists V_b. (G'_b \wedge B' \wedge F_\infty)$ , then

$$F' \wedge B' \neq \emptyset .$$

Since goal containment holds inside  $F_\infty$ ,

$$\begin{aligned} \exists V_b. F_a \wedge (G_b \wedge B \wedge F_\infty) &\subseteq \exists V_b. F'_a \wedge (G'_b \wedge B' \wedge F_\infty) \\ \exists V_b. (F \wedge B \wedge F_\infty) &\subseteq \exists V_b. (F' \wedge B' \wedge F_\infty) \\ \exists V_b. (F \wedge B) &\subseteq \exists V_b. (F' \wedge B') \end{aligned}$$

This is further illustrated in Fig. 5. We call it a *strengthening* because goal containment with  $F_\infty$  increases the chance of getting a proof.

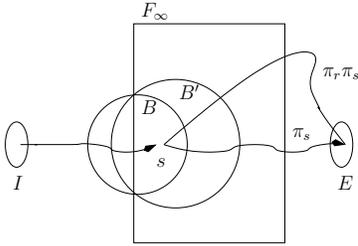


Fig. 5. The strengthened induction condition.

Our use of  $F_\infty$  is similar to the method in [21] on strengthening BMC induction proof with over-approximated reachable states. To ensure termination when computing  $F_\infty$ , we compute exact post-conditions over  $\pi_p\{\pi_r\}^k$  up to a finite set of values of  $k$  and then switch to *widening* [22]. In the working example, for instance,  $F_\infty = i \leq n$ .

### D. Composition of Parameterized Traces

Extending the induction method to handle more complex counterexample patterns is possible. For counterexamples involving concatenation and embedding of recurring segments, we have identified sufficient conditions (special cases) in which goal containment can be checked efficiently. Due to page limit, we omit the discussion on these complex patterns; an extended version of this paper is available upon request.

## VII. EXPERIMENTS

We have implemented the new method in the F-Soft verification platform [23] and integrated with a CEGAR procedure [24]. F-Soft is a tool for analyzing safety properties in C programs by checking whether certain labeled statements

are reachable. It has a preprocessing phase in which complex C statements (such as pointers, arrays, function calls, etc.) are rewritten into simple statements involving scalar variables only. For the simplified C program, it builds a CDFG representation, which is taken as input by subsequent analysis procedures, including CEGAR and our induction based method.

Our test cases are several software benchmarks in the public domain. For each test case, we run both standard CEGAR and the augmented version (with our induction method) of the same CEGAR procedure. All the experiments were conducted on a workstation with 3 GHz Pentium 4 processor and 2GB of RAM running Red Hat Linux 7.2.

### A. The GNU bc Example

Our first test case comes from the GNU *bc* package (*bc-1.06*), which implements a Unix command line calculator with arbitrary precision. There is a known array bounds violation bug in line 176 of the file *storage.c*. The bug is illustrated in the last line of Fig. 6, where the guard (*indx < v\_count*) should be changed to (*indx < a\_count*). This bug is inherently difficult to find with testing [25], since the corrupted heap does not always cause an immediate crash—it often causes a crash when another sometimes unrelated *malloc()* is called.

```

a_count = 256;
...
old_count = a_count;
a_count = a_count + STORE_INCR;
...
arrays = bc_malloc(a_count*sizeof(bc_var_array*));
names = bc_malloc(a_count*sizeof(char*));
...
for (indx=1; indx<old_count; indx++)
    arrays[indx] = old_ary[indx];

for (; indx < v_count; indx++)
    arrays[indx] = NULL;           //failure

```

Fig. 6. Code in *more\_arrays()* of the GNU *bc* example

This bug cannot be detected using standard CEGAR when *a\_count=256*. However, if *a\_count* is set to some small value, standard CEGAR may find a concrete counterexample. In our experiments, *a\_count* is set to various values starting with 1, 2, 3, ... The result is given in Fig. 7. With a small initial value, standard CEGAR found the concrete counterexample, but it demonstrates poor runtime performance and is clearly not scalable.

The CEGAR procedure augmented with our induction method was able to identify and prove the existence of

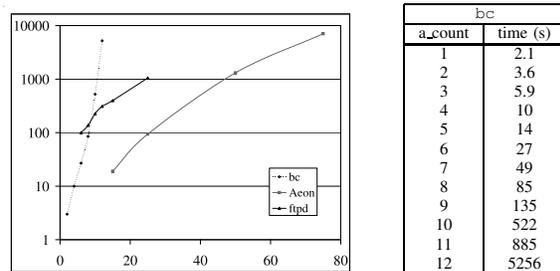


Fig. 7. Run time of the standard CEGAR procedure: x-axis: values of *n*; y-axis: run time in seconds.

a parameterized counterexample within 10 seconds. This is slightly slower than standard CEGAR for `a_count=3`. However, the proof is valid (i.e., a concrete counterexample instance exists) for all `a_count=1, 2, ..., k`.

### B. The Aeon Example

Our second test case comes from the Linux mail transfer agent `Aeon 0.02a`. There is a buffer overflow inside the function `getConfig`, whenever it calls `strcpy` to duplicate a string returned by the function `getenv` to a buffer with size `MAX_LEN`. This bug is representative for many buffer overflows leading to possible security breaches. This example was also studied by Kroening and Weissenbacher in [14].

We applied standard CEGAR as well as the augmented version to this example. When `MAX_LEN=512`, our implementation of standard CEGAR failed to detect the bug within 4 hours (BLAST [26] and SLAM [7] also timed out, as reported in [14]). Our induction based method was able to identify and prove the existence of a parameterized counterexample within 6 seconds. In comparison, the loop detection method in [14] detected the bug within 254.5 seconds; the runtime of their method will keep increasing as `MAX_LEN` becomes larger (its runtime was 25.0 seconds when `MAX_LEN=25`). In contrast, our proof is valid (i.e., a concrete counterexample instance exists) for all `MAX_LEN=1, 2, ..., k`.

### C. The ftpd Example

Our third test case comes from the `wu-ftp-2.6.2` package. There is a buffer overrun inside `ftpprestart.c` when the function `newfile` is called. The induction parameter is `numfiles`, which is 1024 in the concrete counterexample. This example was also studied in [27]. Standard CEGAR failed to detect the bug (although it can find a bug when we set `numfiles` to smaller values, as is shown in Fig. 7), whereas our induction augmented CEGAR procedure found the bug in 22 seconds.

## VIII. CONCLUSIONS

We have presented an induction based method for proving the existence of long counterexamples. The method avoids a state-by-state match of the abstract counterexample during the search for concrete counterexamples. It provides complementary strengths to the popular CEGAR methods. For future work, we want to extend the induction method to handle more complex counterexample patterns.

## REFERENCES

- [1] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in Systems Design*, vol. 19, no. 3, pp. 291–314, 2001.
- [2] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer Aided Design*. Springer, 2000, pp. 108–125, LNCS 1954.
- [3] L. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *Computer Aided Verification (CAV'03)*. Springer, 2003, pp. 1–13, LNCS 2725.
- [4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Mar. 1999, pp. 193–207, LNCS 1579.
- [5] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ: Princeton University Press, 1994.
- [6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification (CAV'00)*. Springer, 2000, pp. 154–169, LNCS 1855.
- [7] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Programming Language Design and Implementation (PLDI'01)*, June 2001, pp. 203–213.
- [8] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification (CAV'03)*. Springer, July 2003, pp. 1–13, LNCS 2725.
- [9] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart simulation using collaborative formal and simulation engines," in *International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2000, pp. 120–126.
- [10] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Programming Language Design and Implementation (PLDI'05)*, June 2005, pp. 213–223.
- [11] K. Nanshi and F. Somenzi, "Guiding simulation with increasingly refined abstract traces," in *Proceedings of ACM/IEEE Design Automation Conference*, 2006, pp. 737–742.
- [12] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi, "Improving Ariadne's bundle by following multiple threads in abstraction refinement," in *International Conference on Computer-Aided Design*, Nov. 2003, pp. 408–415.
- [13] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Design, Automation and Test in Europe (DATE'04)*, Mar. 2004, pp. 10 156–10 161.
- [14] D. Kroening and G. Weissenbacher, "Counterexamples with loops for predicate abstraction," in *Computer Aided Verification (CAV'06)*. Springer, 2006, pp. 152–165, LNCS 4144.
- [15] M. N. Seghir and A. Podolski, "ACSAR: Software model checking with transfinite refinement," in *International SPIN Workshop on Model Checking Software*. Springer, 2007, LNCS 4595.
- [16] T. Ball, O. Kupferman, and M. Sagiv, "Leaping loops in the presence of abstraction," in *Computer Aided Verification (CAV'07)*. Springer, 2007, pp. 491–503, LNCS 4590.
- [17] C. Wang, H. Kim, and A. Gupta, "Hybrid CEGAR: Combining variable hiding and predicate abstraction," in *International Conference on Computer Aided Design (ICCAD'07)*, 2007, to appear.
- [18] Z. Yang, C. Wang, F. Ivančić, and A. Gupta, "Mixed symbolic representations for model checking software programs," in *Formal Methods and Models for Codesign (MEMOCODE'06)*, July 2006, pp. 17–24.
- [19] E. Dijkstra, *A Discipline of Programming*. NJ: Prentice Hall, 1976.
- [20] F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang, "Model checking C program using F-Soft," in *International Conference on Computer Design*, Oct. 2005, pp. 297–308.
- [21] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs complement SAT-based BMC in DiVer," in *Computer Aided Verification (CAV'03)*. Springer, 2003, pp. 206–209, LNCS 2725.
- [22] C. Wang, Z. Yang, A. Gupta, and F. Ivančić, "Using counterexamples for improving the precision of reachability computation with polyhedra," in *Computer Aided Verification (CAV'07)*. Springer, 2007, pp. 352–265, LNCS 4590.
- [23] F. Ivančić, Z. Yang, I. Shlyakhter, M. Ganai, A. Gupta, and P. Ashar, "F-SOFT: Software verification platform," in *Computer-Aided Verification*. Springer, 2005, pp. 301–306, LNCS 3576.
- [24] H. Jain, F. Ivančić, A. Gupta, I. Shlyakhter, and C. Wang, "Using statically computed invariants inside the predicate abstraction and refinement loop," in *Computer Aided Verification (CAV'06)*. Springer, 2006, pp. 137–151, LNCS 4144.
- [25] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *International Conference on Machine Learning*, 2006, pp. 1105–1112.
- [26] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Principles of programming languages (POPL'02)*, 2002, pp. 58–70.
- [27] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek, "Buffer overrun detection using linear programming and static analysis," in *ACM Conference on Computer and Communications Security*, Oct. 2003, pp. 345–354.

# Lifting Propositional Interpolants to the Word-Level

Daniel Kroening  
Computer Systems Institute  
ETH Zurich

Georg Weissenbacher  
Computer Systems Institute  
ETH Zurich

**Abstract**—Craig interpolants are often used to approximate inductive invariants of transition systems. Arithmetic relationships between numeric variables require word-level interpolants, which are derived from word-level proofs of unsatisfiability. While word-level theorem provers have made significant progress in the past few years, competitive solvers for many logics are based on flattening the word-level structure to the bit-level. We propose an algorithm that lifts a resolution proof obtained from a bit-flattened formula up to the word-level, which enables the computation of word-level interpolants. Experimental results for equality logic suggest that the overhead of lifting the propositional proof is very low compared to the solving time of a state-of-the-art solver.

## I. INTRODUCTION

Fifty years ago, William Craig showed that for each inconsistent pair of logical formulas  $\langle A, B \rangle$ , there exists a formula  $\phi$  – the *Craig interpolant* – that is implied by  $A$ , inconsistent with  $B$ , and refers only to non-logical symbols common to  $A$  and  $B$  [1]. Intuitively, the interpolant  $\phi$  can be understood as an abstraction of  $A$ . This result has been recently rediscovered and is the basis of various abstraction techniques in several automated verification tools. All of these verification techniques require an efficient decision procedure that is able to generate interpolants for unsatisfiable formulas. In program verification, the queries that typically arise are stated in quantifier-free Presburger arithmetic, equality- or difference-logic. These theories enjoy the *small-model* property, i.e., if a formula is satisfiable, then it has a satisfying assignment in a finite domain. Many decision procedures exploit this property and translate the original problem (either eagerly or lazily) into propositional logic. These instances can then be solved efficiently due to the recent advances in Boolean satisfiability solving [2]. The disadvantage of this approach is the loss of structure: Even though interpolants can be easily extracted from the resolution proofs provided by proof logging SAT solvers [3], it is prohibitively complicated to use the resulting bit-level interpolants in combination with word-level implementations or specifications.

**Contribution:** While interpolating decision procedures that generate word-level interpolants do exist [4], they are not yet competitive with state-of-the-art SAT-based theorem provers. We solve this problem by *lifting* existing resolution proofs generated by a SAT-based decision procedure to *word-level proofs*, from which the corresponding word-level inter-

polants can be easily extracted. Exemplarily, we provide a proof-lifting algorithm for equality logic.

## Related Work

Our algorithm is not the first that constructs word-level interpolants: McMillan’s theorem prover FOCI generates interpolants for quantifier free formulas with linear inequalities and uninterpreted function symbols [4]. His approach requires a tailor-made theorem prover for these theories. The technique presented in this paper is suitable for off-the-shelf, bit-level decision procedures, which are known to perform very well on a variety of logics, e.g., equality logic and bit-vector arithmetic.

Interpolants have various applications in software and hardware model checking. For instance, interpolation can be used to derive an abstract image operator from failed attempts to disprove a property of a finite state transition system. By computing a fix-point for this operator, one can obtain an inductive invariant of the transition system [5].

## II. BACKGROUND

### A. Propositional Craig Interpolants

A propositional formula consists of *atoms* (indicated by Boolean identifiers  $a_i$ ,  $i = 1, 2, 3, \dots$ ), the constants **true** and **false**, and the operators  $\wedge$ ,  $\vee$ , and  $\neg$ . We use  $\bar{a}_i$  as an alternative notation for  $\neg a_i$ . A propositional formula is either an atom, or of the form  $(F \wedge G)$ ,  $(F \vee G)$  or  $\bar{F}$ , where  $F$  and  $G$  are also propositional formulas. We write  $\mathcal{A}(F)$  to denote the set of atoms that occur in  $F$ . We use  $\Sigma$  to denote a valuation to  $\mathcal{A}(F)$ .  $\Sigma(a_i)$  denotes the truth value of the atom  $a_i$  in  $\Sigma$ , and  $\Sigma(a_i) = \perp$  denotes that  $a_i$  is not assigned by  $\Sigma$ .  $F$  is tautological if it evaluates to true for *all* Boolean valuations of  $\mathcal{A}(F)$ , and unsatisfiable if there is no such valuation.

A *literal* is either an atom or the negation of an atom. A disjunction of literals is called a *clause*. In clauses, we sometimes omit the disjunction operator  $\vee$ , e.g., we write  $(\bar{a}_1 \bar{a}_2 a_3)$  instead of  $(\bar{a}_1 \vee \bar{a}_2 \vee a_3)$ . Furthermore, we use  $\square$  to denote the empty clause. A propositional formula of the form  $F = \bigwedge_{i=1}^n \left( \bigvee_{j=1}^m L_{i,j} \right)$ , with  $L_{i,j} \in \{a_1, a_2, \dots\} \cup \{\bar{a}_1, \bar{a}_2, \dots\}$ , is in conjunctive normal form (CNF). For each propositional formula, there exists a logically equivalent formula in CNF.

Given two clauses  $a_i \vee \Theta$  and  $\bar{a}_i \vee \Theta'$  (where  $\Theta$  and  $\Theta'$  represent disjunctions of literals), the *resolvent* of these clauses is the clause  $\Theta \vee \Theta'$ . The corresponding proof rule is known as *resolution*:

$$\text{RES} \frac{a_i \vee \Theta \quad \bar{a}_i \vee \Theta'}{\Theta \vee \Theta'} \quad (1)$$

This research was supported by Microsoft Research through its European PhD Scholarship Programme, by SRC contract no. 2006-TJ-1539, and by an award from IBM Research.

A propositional formula  $F$  in CNF is unsatisfiable iff the empty clause can be derived by a sequence of such resolution steps starting from the clauses of  $F$ .

A sequence of resolution steps can be represented by a directed acyclic graph, where each vertex  $c$  with no predecessor is a clause of  $F$ , and all other vertices are resolvents of their (exactly two) predecessors. Such a graph is a proof of unsatisfiability if its root node is the empty clause.

Given an unsatisfiable formula  $F$  partitioned into two sets of clauses  $\langle A, B \rangle$ , an interpolant for  $A$  and  $B$  is a formula  $P$  such that

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

An interpolant for an unsatisfiable set of clauses  $\langle A, B \rangle$  can be derived from a resolution proof in linear time [3], [4].

### B. Propositional Encodings of Decision Problems

In logics that are more expressive than propositional logic (e.g., equality logic), formulas may contain atoms that are specific to the theory that is used. We call these atoms the *Theory Atoms*. Let  $\mathcal{A}^T(\varphi) \subseteq \mathcal{A}(\varphi)$  denote the set of Theory Atoms in  $\varphi$  that are not Boolean identifiers or constants.

*Definition 1 (Propositional Skeleton):* The *Propositional Skeleton* of  $\varphi$  is denoted by  $\varphi_{sk}$  and is obtained by replacing every theorem atom  $\alpha \in \mathcal{A}^T(\varphi)$  by a new Boolean identifier. The Boolean identifier that replaces  $\alpha$  is denoted by  $e_\alpha$ .

We simply write  $\mathcal{A}^T$  for  $\mathcal{A}^T(\varphi)$  if the formula  $\varphi$  is clear from the context.

*Definition 2 (Corresponding Constraint):* Given a truth assignment  $\Sigma : \mathcal{A}^T \rightarrow \{\text{true}, \text{false}, \perp\}$  to the theory atoms of formula  $\varphi$ , we define the *corresponding constraint*  $\Psi(\Sigma)$  as the conjunction of the theory atoms  $\alpha \in \mathcal{A}$  with  $\Sigma(\alpha) \neq \perp$ , where the theory atoms  $\alpha$  with  $\Sigma(\alpha) = \text{false}$  are negated:

$$\Psi(\Sigma) := \bigwedge_{\alpha \in \mathcal{A}^T | \Sigma(\alpha) \neq \perp} \begin{cases} \alpha & : \Sigma(\alpha) = \text{true} \\ \neg \alpha & : \Sigma(\alpha) = \text{false} \end{cases}$$

## III. INTERPOLATION FOR EQUALITY LOGIC

### A. Flattening Equality Logic

We describe how to lift propositional proofs for equality logic to the word level. Equality logic permits formulas with an arbitrary Boolean structure, but limits atoms to Boolean variables and equalities of the form  $x = y$ , where  $x, y$  are variables or numeric constants over some infinite domain  $D$ .<sup>1</sup> We call these non-Boolean variables the *theory variables*, and denote the formula by  $\varphi$ .

Theorem provers for equality logic and uninterpreted functions are reasonably efficient, and usually rely on a combination of a propositional SAT solver and an implementation of *Congruence Closure*, which is based on the union-find algorithm. They compute a propositional encoding in a lazy

<sup>1</sup>A construction of a word-level proof that uses only the tree proof rules for equality given in Sec. III-D from a bit-level refutation is not always possible for a finite domain.

manner. Nevertheless, algorithms that perform *range allocation* for equality logic are still superior to procedures that compute encodings lazily.

The idea of range allocation is to compute a bounded range of integer values for the theory variables in  $\varphi$ . This range is constructed such that it is sufficient for a satisfying assignment of  $\varphi$ , if one exists. We restrict the presentation to ranges that can be encoded with bit-vectors, i.e., values from 0 to  $2^n - 1$  for some integer  $n$ .

*Definition 3 (Range):* Let  $V$  denote the set of theory variables in  $\varphi$ . A *range*  $R : V \rightarrow \mathbb{N}$  is an assignment of a number of bits to each variable in  $V$ . We denote the Boolean variable that encodes bit  $i \in \{0, \dots, R(x) - 1\}$  for  $x \in V$  by  $x_i$ . These Boolean variables are called *vector variables*.

We assume that the range is *consistent*, i.e., for any equality  $x = y$  in  $\varphi$ ,  $R(x) = R(y)$  holds.

*Definition 4 (Small-Domain Assignment):* An assignment  $\sigma$  to the variables  $V$  with  $\sigma(x) \in \{0, \dots, 2^{R(x)} - 1\}$  is called a *small-domain assignment*.

*Definition 5 (sufficient):* A range is *sufficient* for a formula  $\varphi$  if there is a small-domain assignment  $\sigma$  for every satisfiable corresponding constraint  $\Psi(\Sigma)$  (see Def. 2).

Given a consistent range, the variables are interpreted as bit-vectors with just enough bits to encode the values in the range. Formally, we denote the propositional constraint for  $x = y$ , where  $x$  and  $y$  are non-Boolean variables, by  $E(x = y)$ . Let  $n = R(x) = R(y)$ . The propositional constraint is defined as follows:

$$E(x = y) : \iff e_{x=y} \iff \bigwedge_{i=0}^{n-1} x_i \iff y_i \quad (2)$$

Recall that  $e_{x=y}$  is the variable used to replace  $x = y$  in the skeleton  $\varphi_{sk}$ . We assume that the constants in the formula are mapped to the range. The propositional encoder for an equality between a variable and a constant is straight-forward.

*Lemma 1:* Given a sufficient range,  $\varphi$  and

$$\varphi_{sk} \wedge \bigwedge_{\alpha \in \mathcal{A}^T} E(\alpha) \quad (3)$$

are equi-satisfiable. We denote formula (3) as  $\varphi_{enc}$ .

We assume that a sufficient range is given. An obviously sufficient range is  $R(v) = \lceil \log_2 |V| \rceil$  [6]. Procedures to compute a smaller but still sufficient range are beyond the scope of this article. The techniques we propose are still applicable even if smaller ranges are used as long as the range is sufficient according to Def. 5.

*Example 1:* Consider the formula

$$x = y \wedge y = z \wedge z \neq x$$

as a running example. A sufficient range is  $R(x) = R(y) = R(z) = 2$ , which results in the following propositional encoding:

$$\begin{array}{ll} e_{x=y} & \iff (x_0 \iff y_0 \wedge x_1 \iff y_1) & \text{for } x = y, \\ \wedge & e_{y=z} & \iff (y_0 \iff z_0 \wedge y_1 \iff z_1) & \text{for } y = z, \\ \wedge & e_{z=x} & \iff (z_0 \iff x_0 \wedge z_1 \iff x_1) & \text{for } z = x, \\ \wedge & & e_{x=y} \wedge e_{y=z} \wedge \overline{e_{z=x}} & \text{for } \varphi_{sk} \end{array}$$

## B. Bit-Level Resolution Proofs for Equality Logic

The propositional formula  $\varphi_{enc}$  can be converted into CNF, and is then passed to a propositional SAT solver. In case  $\varphi$  is unsatisfiable, so is  $\varphi_{enc}$ , and we can obtain a propositional resolution proof from the SAT solver. The form of this proof depends on the particular encoding that is used to convert  $\varphi_{enc}$  into CNF. We define a particular encoding in order to record claims about the propositional resolution proofs that are obtained. However, the method is not limited to a particular encoding.

We restrict the presentation to CNF obtained by means of Tseitin's encoding [7], which is the basic technique behind most tools that generate CNF.

*Definition 6 (Bit-Flattening of E):* The propositional constraint  $E(x = y)$  is transformed into CNF using auxiliary variables  $o_0, \dots, o_{n-1}$ , where  $o_i$  holds if  $x_i \longleftrightarrow y_i$ . Note that these auxiliary variables are specific to  $E(x = y)$ , and are not shared with other constraints.

$$\begin{aligned} E(x = y) &\iff \\ &\bigwedge_{i=0}^{n-1} (\overline{x_i y_i o_i}) \wedge (x_i y_i o_i) \wedge (\overline{x_i y_i \overline{o_i}}) \wedge (x_i y_i \overline{o_i}) \\ &\wedge \bigwedge_{i=0}^{n-1} (o_i \overline{e_{x=y}}) \wedge (\overline{o_0} \dots \overline{o_{n-1}} e_{x=y}) \end{aligned} \quad (4)$$

This encoding may be optimized in numerous ways, e.g., many clauses can be omitted if the polarity of the atoms is exploited. These techniques are beyond the scope of this article, but our method is still applicable after applying the commonly used optimizations. The propositional skeleton  $\varphi_{sk}$  is transformed by similar means, i.e., introducing a new auxiliary variable for each node of the parse-tree of  $\varphi_{sk}$ .

*Example 2:* Continuing our running example from the previous subsection, the following set of clauses<sup>2</sup> is a CNF encoding of the formula  $\varphi_{enc}$ :

$$\begin{aligned} &\text{for } x = y: \\ &x_0 \overline{y_0} \overline{o_{10}} \quad \overline{x_0} y_0 \overline{o_{10}} \quad x_1 \overline{y_1} \overline{o_{11}} \quad \overline{x_1} y_1 \overline{o_{11}} \quad o_{10} \overline{e_{x=y}} \quad o_{11} \overline{e_{x=y}} \\ &\text{for } y = z: \\ &y_0 \overline{z_0} \overline{o_{20}} \quad \overline{y_0} z_0 \overline{o_{20}} \quad y_1 \overline{z_1} \overline{o_{21}} \quad \overline{y_1} z_1 \overline{o_{21}} \quad o_{20} \overline{e_{y=z}} \quad o_{21} \overline{e_{y=z}} \\ &\text{for } z = x: \\ &x_0 z_0 o_{30} \quad \overline{x_0} \overline{z_0} o_{30} \quad x_1 z_1 o_{31} \quad \overline{x_1} \overline{z_1} o_{31} \quad \overline{o_{30} o_{31}} e_{z=x} \\ &\text{for } \varphi_{sk}: \\ &e_{x=y} \quad e_{y=z} \quad \overline{e_{z=x}} \end{aligned}$$

The variables  $o_{10}$ ,  $o_{11}$ , and so on are auxiliary variables added for Tseitin's encoding of the conjunction required for decomposing the equalities.

## C. Lifting Bit-Level Proofs for Equality Logic

We represent propositional resolution proofs  $P$  as a binary tree, where the nodes represent the clauses that were resolved from their antecedent nodes using the RES rule (see (1) in Section II-A). For instance, Figure 1(a) shows the resolution

<sup>2</sup>In order to simplify the presentation, the clauses that are trivially satisfied after propagation of the unit-clauses  $e_{x=y}$ ,  $e_{y=z}$ , and  $\overline{e_{z=x}}$  are omitted.

tree for the set of clauses given in Example 2. We write  $root(P)$  for the root-node of  $P$ . We write  $leaf(n)$  if  $n$  is a leaf node. Otherwise, we write  $L(n)$  and  $R(n)$  for the left and right antecedent nodes, respectively. We write  $\mathcal{L}(n)$  to denote the fact at node  $n \in P$  (the *label* of node  $n$ ). For leaf-nodes that are generated for a specific atom (as opposed to the skeleton), we write  $\mathcal{A}^T(n)$  to denote the theory atom that the clause was generated for. A proof  $P$  shows unsatisfiability of a formula  $\varphi$  if its leaves are trivially implied by  $\varphi$  and  $\mathcal{L}(root(P)) = \square$ . We write  $\varphi \vdash_P \square$  in this case.

Our proof-lifting algorithms take a propositional resolution proof  $P$  for  $\varphi_{enc}$  as input, and lift it up to a new proof for  $\varphi$  that uses a word-level logic by replacing the labels of the nodes. The structure of the graph is not changed, up to a final minimization step.

For the theory of equality and the encoding described above, the lifted propositions have one of the following three forms, where  $T_i$  denotes a theory predicate:

- F1  $T_1 \wedge \dots \wedge T_k$ , or
- F2  $T_1 \wedge \dots \wedge T_k \vee \Theta$ , where  $\Theta$  is a propositional clause, or
- F3 a propositional CNF-clause, including the empty clause.

We define the following *lifting-function*  $\lambda(n)$  for the propositions that are used to label the leaf-nodes of  $P$  by means of a case-split on the clause  $\mathcal{L}(n)$ .

$$\lambda(n) := \begin{cases} x \neq y \vee o_i & : \text{ for } \mathcal{L}(n) = (\overline{x_i y_i} o_i) \\ x \neq y \vee o_i & : \text{ for } \mathcal{L}(n) = (x_i y_i o_i) \\ x = y \vee \overline{o_i} & : \text{ for } \mathcal{L}(n) = (\overline{x_i y_i} \overline{o_i}) \\ x = y \vee \overline{o_i} & : \text{ for } \mathcal{L}(n) = (x_i y_i \overline{o_i}) \\ \mathcal{L}(n) & : \text{ otherwise} \end{cases} \quad (5)$$

We note that the clauses generated by  $\lambda(n)$  for leaf-nodes have either one of the forms F2 or F3. It is easy to see that the label returned by  $\lambda$  is trivially implied by  $\varphi$  for all leaf-nodes  $n$  of  $P$ .

The inner nodes of  $P$  (i.e., those that are the result of a resolution step) are also lifted. Let

$$\begin{aligned} \lambda(L(n)) &= T_1^L \wedge \dots \wedge T_j^L \vee \Theta, \quad \text{and} \\ \lambda(R(n)) &= T_1^R \wedge \dots \wedge T_k^R \vee \Theta' \end{aligned}$$

be the lifted labels of the two antecedent nodes of  $n$ , where  $\Theta$  and  $\Theta'$  stand for (possibly empty) propositional clauses. We define  $T^n$  to denote the conjunction

$$T_1^L \wedge \dots \wedge T_j^L \wedge T_1^R \wedge \dots \wedge T_k^R.$$

There are two cases, depending on whether the resolution that results in node  $n$  is performed on a vector variable:

- If the resolution for node  $n$  is performed by resolving on a vector variable, we define  $\lambda(n) := T^n \vee \Theta \vee \Theta'$  if  $T^n$  is satisfiable, and  $\lambda(n) := \Theta \vee \Theta'$  otherwise.
- Otherwise, the resolution for node  $n$  is performed on a non-vector variable. Let  $\text{RES}(\Theta, \Theta')$  denote the resolvent of  $\Theta$  and  $\Theta'$  according to Rule (1) in Section II-A. Then,  $\lambda(n) := T_1^L \wedge \dots \wedge T_j^L \vee T_1^R \wedge \dots \wedge T_k^R \vee \text{RES}(\Theta, \Theta')$ .

Note that the second transformation may violate the assumption that the lifted clauses are always of form F1, F2, or F3.

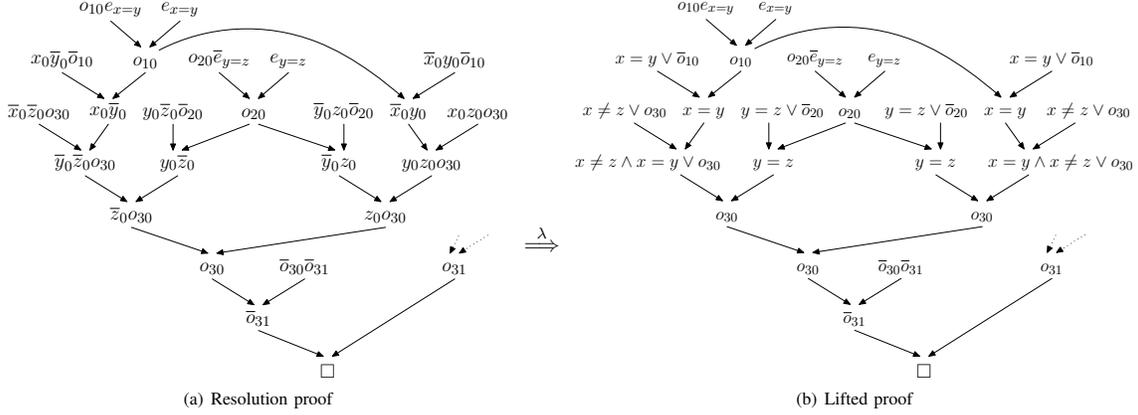


Fig. 1. Resolution graph and the corresponding lifted proof for the clauses in Example 2. The right part of the respective graphs is symmetric to the left part and has therefore been omitted.

Therefore, it is only performed if either  $T_1^L \wedge \dots \wedge T_j^L$  is equal to  $T_1^R \wedge \dots \wedge T_k^R$ , or at least one of these terms is false. In all other cases, the lifting of the propositional proof fails. This case occurs in none of our benchmarks (see Section IV). The satisfiability of  $T^n$  can be checked efficiently using a union-find data structure.

*Theorem 1:* The new labels  $\lambda(n)$  imply the old labels  $\mathcal{L}(n)$ .

*Proof:* (By induction) For the leaf nodes, one can easily show that  $\lambda(n)$  implies  $\mathcal{L}(n)$  by flattening the theory atom in  $\mathcal{L}(n)$  according to Definition 6. This constitutes the base case.

It remains to show that  $\lambda(n)$  implies  $\mathcal{L}(n)$  for an inner node  $n$ . By our induction hypothesis,  $\lambda(L(n))$  implies  $\mathcal{L}(L(n))$  (and  $\lambda(R(n)) \Rightarrow \mathcal{L}(R(n))$ , respectively). Observe that the lifting function  $\lambda$  preserves the propositional structure of  $\Theta$ , the clause over the non-vector variables: If  $\mathcal{L}(L(n))$  is  $\bigvee_{i=1}^k L_i^L \vee \Theta$ , where each  $L_i^L$  denotes a literal for a vector variable, and  $\Theta$  contains no vector variables, then  $\lambda(L(n))$  is of the form  $T_1^L \wedge \dots \wedge T_j^L \vee \Theta$ .

First we consider the case that resolution is performed on a vector variable. We prove

$$(T_1^L \wedge \dots \wedge T_j^L) \wedge (T_1^R \wedge \dots \wedge T_k^R) \vee (\Theta \vee \Theta') \Rightarrow \text{RES}\left(\bigvee_{i=1}^l L_i^L, \bigvee_{i=1}^m L_i^R\right) \vee (\Theta \vee \Theta') \quad (6)$$

by performing a case split over values of the elements of the disjunction on the left side of the implication: If  $(\Theta \vee \Theta')$  is true, or the expression on the left side is false, then the implication holds trivially. In the remaining case,  $\Theta \vee \Theta'$  is false, and  $(T_1^L \wedge \dots \wedge T_j^L) \wedge (T_1^R \wedge \dots \wedge T_k^R)$  holds. Then  $T_1^L \wedge \dots \wedge T_j^L$  implies  $\bigvee_{i=1}^l L_i^L$ , and  $T_1^R \wedge \dots \wedge T_k^R$  implies  $\bigvee_{i=1}^m L_i^R$  by our induction hypothesis, and (6) holds.

For the remaining case (i.e., when we perform resolution on a non-vector variable), we have to show that

$$(T_1^L \wedge \dots \wedge T_j^L) \vee (T_1^R \wedge \dots \wedge T_k^R) \vee \text{RES}(\Theta, \Theta') \Rightarrow \bigvee_{i=1}^l L_i^L \vee \bigvee_{i=1}^m L_i^R \vee \text{RES}(\Theta \vee \Theta')$$

This is trivial, since the new label  $\lambda(n)$  as well as the old label  $\mathcal{L}(n)$  is obtained from the predecessors of  $n$  by resolution. ■

It is left to show that the lifted labeling actually corresponds to a proof.

*Theorem 2:* For all inner nodes  $n$  of  $P$ , the new label is implied by the conjunction of the new antecedent labels:

$$\lambda(L(n)) \wedge \lambda(R(n)) \Rightarrow \lambda(n) \quad (7)$$

*Proof:* In the case that the resolution is performed on a vector variable, we have to show that

$$\left( (T_1^L \wedge \dots \wedge T_j^L) \vee \Theta \right) \wedge \left( (T_1^R \wedge \dots \wedge T_k^R) \vee \Theta' \right) \Rightarrow (T_1^L \wedge \dots \wedge T_j^L) \wedge (T_1^R \wedge \dots \wedge T_k^R) \vee \Theta \vee \Theta'$$

holds. This can be easily achieved by applying the distributive law to the left side of the implication.

The remaining case is trivial, since the  $\lambda(n)$  is obtained by performing resolution on the labels of  $L(n)$  and  $R(n)$ . ■

The lifting-function  $\lambda$  can be applied recursively to a proof  $P$  together with labels  $\mathcal{L}$ , beginning with the leaf-nodes, to produce a new set of labels. We write  $\lambda(P)$  for this new proof.

*Theorem 3:* Let  $P$  denote a proof with  $\varphi_{enc} \vdash_P \square$ , let  $\lambda$  denote a lifting-function, and  $P' := \lambda(P)$  a proof generated by lifting  $P$  using  $\lambda$ . If the lifting function has the following properties, then  $P'$  shows unsatisfiability of  $\varphi$ :

- 1) The lifted leaf-node labels are implied by  $\varphi$ ,
- 2) the lifted inner node labels imply the old labels.

*Proof:* Observe that  $\mathcal{L}(\text{root}(P)) = \square$ . Because of the second premise,  $\lambda(\text{root}(P)) = \square$ . As the leaves  $\lambda(\text{leaf}(n))$  are implied by  $\varphi$ , and  $P'$  is a proof, we have  $\varphi \vdash_{P'} \square$ . ■

#### D. Interpolation using Lifted Proofs

In this section we explain how an interpolant can be derived from the lifted proof  $\lambda(P)$ . Note that the all lifted labels are of form  $F1$ ,  $F2$  or  $F3$  (see Section III-C). Our implementation guarantees that the formula  $\mathcal{P}(T_1, \dots, T_k)$  over the theory predicates is always of form  $T_1 \wedge \dots \wedge T_k$ .

The decision procedure for equality logic (as presented in [4]) makes use of transitivity, reflexivity, and contradiction.

The following two rules are sufficient to decide pure equality logic:

$$\text{TRANS} \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3} \quad \text{EQNEQ} \frac{t_1 = t_2}{\text{false}} \neg(t_1 = t_2)$$

If the formula is unsatisfiable, then a contradictory fact  $(t_i = t_j) \wedge (t_i \neq t_j)$  can be inferred for some  $i, j$ .

A chain of equalities  $(x = t_1) \wedge (t_1 = t_2) \wedge \dots \wedge (t_n = y)$  can be partitioned into maximal sub-chains  $(t_i = t_{i+1}) \dots (t_{j-1} = t_j)$ ,  $i < j$  consisting of either only equalities from  $A$ , or only equalities from  $B$ . Each subchain can then be summarized by  $(t_i = t_j)$ . We maintain such summaries for each node of the lifted proof. By constructing summaries from  $A$  such that the terms  $t_i$  and  $t_j$  also occur in  $B$  (i.e., are *global*) one can obtain an interpolant for  $A$  and  $B$ .

*Example 3:* Consider the lifted proof in Figure 1(b), and the partitioning  $A = (x = y) \wedge (y = z)$  and  $B = (z \neq x)$ . The variable  $y$  is local to  $A$ , while  $x$  and  $z$  are global.

Consider the resolvent of  $x \neq z \wedge x = y \vee o_3 0$  and  $y = z$ . Using transitivity, we derive

$$\text{TRANS} \frac{x = y \quad y = z}{x = z} \{(x = y), (y = z)\} \subseteq A$$

and obtain a maximal chain  $x = z$  as summary for  $A$ . We obtain a contradiction by applying the EQNEQ rule:

$$\text{EQNEQ} \frac{x = z}{\text{false}} \neg(z = x) \in B$$

This yields  $(x = z)$  as an interpolant for  $(x = y) \wedge (y = z) \wedge (z \neq x)$ . We apply the rules presented in [4] to the remaining propositional resolution steps of the lifted proof. Since the propositional structure of  $\varphi$  contains no disjunction, this results in  $(x = z)$  as the interpolant for  $\langle A, B \rangle$ .

#### IV. EXPERIMENTAL RESULTS

We have implemented the techniques described above in a tool called LIFTER. It uses MiniSat [2] as the solver for the encoding  $\varphi_{enc}$ .

We compare the performance of LIFTER with the SMT solver Yices [8] and with FOCI [4]. Of these provers, only FOCI and LIFTER are interpolating. We nevertheless include Yices in the comparison as a reference point.<sup>3</sup>

The benchmarks we use have been suggested in [6], [9], and are available for download. We remove ‘easy’ benchmarks from the set by eliminating a benchmark if all solvers are able to solve it within 0.1s or less. All formulas we consider are unsatisfiable. For the purpose of computing an interpolant, we split the formulas into two parts. We plot the run-time of LIFTER and Yices in Fig. 2. We omit the comparison with FOCI, as Yices outperforms FOCI on most of these queries. The experiments show that on this set of benchmarks, the range encoding is typically faster than the DPLL(T)-based solver. The run-time of LIFTER is dominated by the time taken to build  $\varphi_{enc}$  and transform it into CNF. Both the time taken by

<sup>3</sup>Yices won the most recent SMT competition in all divisions. We also ran CVC3 on all benchmarks, and found that Yices always outperforms CVC3. We therefore do not include CVC3 in the comparison.

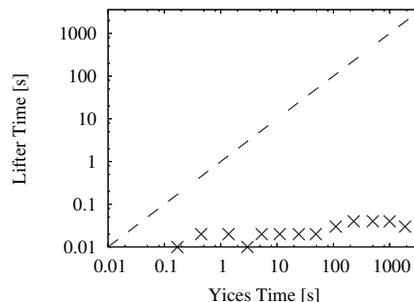


Fig. 2. Comparison of LIFTER and Yices on Equality Logic formulas

MiniSat and the lifting-procedure are negligible. The lifting-procedure was always able to generate a word-level proof.

#### V. CONCLUSION

We present a procedure for constructing a word-level proof for equality logic. We do not rely on a deductive engine to build a proof for the original formula, but instead transform an existing bit-level proof to the word-level. Thus, the deductive engine only has to work on a restricted set of theory-literals, and does not perform any case-splitting. From the resulting proofs, word-level Craig interpolants can be obtained. The procedure generates a word-level interpolant for all of our equality logic benchmarks. As future work, we plan to lift proofs obtained from highly optimized incremental encodings for bit-vector arithmetic, e.g., the one described in [10].

#### ACKNOWLEDGMENTS

We would like to thank Angelo Brillout and Christoph Wintersteiger for their helpful comments on this paper.

#### REFERENCES

- [1] W. Craig, “Linear reasoning. A new form of the Herbrand-Gentzen theorem.” *Journal of Symbolic Logic*, vol. 22, no. 3, pp. 250–268, 1957.
- [2] N. Eén and N. Sörensson, “An extensible SAT-solver.” in *Theory and Applications of Satisfiability Testing (SAT)*, 2003, pp. 502–518.
- [3] P. Pudlák, “Lower bounds for resolution and cutting plane proofs and monotone computations,” *The Journal of Symbolic Logic*, vol. 62, no. 3, pp. 981–998, 1997.
- [4] K. L. McMillan, “An interpolating theorem prover,” *Theoretical Computer Science*, vol. 345, no. 1, pp. 101–121, 2005.
- [5] —, “Interpolation and SAT-based model checking,” in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 2725. Springer, 2003, pp. 1–13.
- [6] B. Dutertre and L. de Moura, “The small model property: How small can it be?” *Information and Computation*, vol. 178, no. 1, pp. 279–293, 2002.
- [7] G. Tseitin, “On the complexity of proofs in propositional logics,” in *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, J. Siekmann and G. Wrightson, Eds., vol. 2. Springer, 1983, originally published 1970.
- [8] B. Dutertre and L. de Moura, “The Yices SMT solver,” Available at <http://yices.csl.sri.com/tool-paper.pdf>, 2006.
- [9] Y. Rodeh and O. Strichman, “Building small equality graphs for deciding equality logic with uninterpreted functions.” *Information and Computation*, vol. 204, no. 1, pp. 26–59, 2006.
- [10] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady, “Deciding bit-vector arithmetic with abstraction,” in *Tools and Algorithms for the construction and Analysis of Systems (TACAS)*, ser. LNCS, vol. 4424. Springer, 2007, pp. 358–372.

# Global Optimization of Compositional Systems

Fadi Zaraket

John Pape

Adnan Aziz

Magarida Jacome

Sarfraz Khurshid

The University of Texas at Austin

**Abstract**—Embedded systems typically consist of a composition of a set of hardware and software IP modules. Each module is heavily optimized by itself. However, when these modules are composed together, significant additional opportunities for optimizations are introduced because only a subset of the entire functionality is actually used. We propose COSE—a technique to jointly optimize such designs. We use symbolic execution to compute invariants in each component of the design. We propagate these invariants as constraints to other modules using global flow analysis of the composition of the design. This captures optimizations that go beyond, and are qualitatively different than, those achievable by compiler optimization techniques such as common subexpression elimination, which are localized. We again employ static analysis techniques to perform optimizations subject to these constraints. We implemented COSE in the Metropolis platform and achieved significant optimizations using reasonable computational resources.

## I. INTRODUCTION

An embedded system performs one or more dedicated tasks using one or more processors that communicate with other dedicated devices, without being referred to as a computer [1], [2]. Embedded systems arise in many applications: examples include communications, image processing, and automotive electronics [2].

Embedded systems are heterogeneous by nature. For example, they combine application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) with embedded software. A diversity of platforms try to employ and integrate different models of computation (MoC) to address the co-design problem of embedded systems. The platforms need to provide tools to specify, synthesize, and validate an embedded design.

Metropolis is an example of a design tool for embedded systems. The designer specifies the system using the Metropolis Metamodel (MMM) and Metropolis synthesizes the design by configuring parametrized architectural elements, dynamic scheduling algorithms, interface blocks, as well as final software and hardware implementation. Many techniques exist to optimize each of these components as well as the underlying communication network [3], [4], [5]. However, there is a need for developing new techniques that can perform optimizations across hierarchical boundaries.

We present *co-optimization using symbolic execution* (COSE), a novel technique that works across components of an embedded design to optimize structures therein. COSE automatically detects opportunities for optimization thereby avoiding labor on the part of the designer who would otherwise have to manually select the subset of functionality used. COSE detects invariants that the designer may not know or may not recognize as useful for optimizing other components. Furthermore, there is anecdotal evidence that having a designer

manually simplify components using knowledge of their environments is a buggy process; we use a formal technique to compute opportunities for optimization, thereby guaranteeing the optimized and original designs are functionally equivalent.

Key to our approach is symbolic execution (SE) [6], [7]. We use SE to analyze software components and define a limited set of values that software feeds hardware as constraints. SE explores all possible paths of execution of the code specifying a component. It does this by accumulating *path conditions* (PCs) and annotating them to the corresponding segments of the component. A PC is associated with a branch of code and consists of the conjunction of conditions over input and state variables necessary and sufficient for the branch to execute. These PCs define constraints that limit the set of values that software feeds hardware. We propagate these constraints across the networks of the design and use *static analysis* techniques such as *constant propagation*, *redundancy removal*, and *don't care* optimizations to reduce the hardware components.

SE performs particularly well in the context of simple inequality checks and mapping assignments. This makes it suitable for our problem of detecting invariants that raise from different configurations and parameterizations of hardware components in software. COSE applies Juzi [6], an SE tool, to extract PCs for every line of code in an MMM component. Juzi uses CVC-lite [8] to solve or simplify PCs. CVC-Lite is a theorem prover for the Satisfiability Modulo Theory problem and it operates on logics that can express equality, inequality, and arithmetic operations.

We make the following key contributions:

- 1) We develop COSE—a technique that crosses software and hardware boundaries to perform joint optimizations.
- 2) We use SE to detect opportunities for optimization that are qualitatively different from those detectable with localized compiler optimizations and computationally infeasible with pure Boolean netlist analysis.
- 3) We implemented a prototype for COSE in the Metropolis framework and achieved significant results on two designs with significant complexity—a realtime image processing system and a switch fabric supporting mixed-mode traffic.

Our paper is structured as follows: we present a motivating example in Section II. In Section III-A we introduce Metropolis and the MMM. We illustrate SE with an example and elaborate on its advantages and limitations in Section III-B. In Section IV we present our prototype implementation. We present two case studies in Section V. We discuss related work in Section VI, and we finally conclude in Section VII.

We convey the key ideas behind COSE using examples and high-level description and rely on the Metropolis metamodel

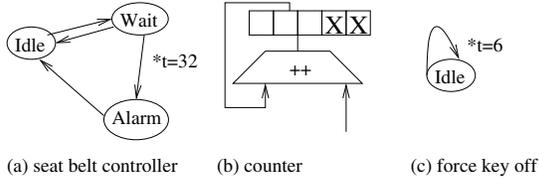


Fig. 1. Modified seat belt

to formalize the different techniques we use as well as their compatibility. We use the informal style for pedagogical reasons, as well as the 8 page submission limit.

## II. EXAMPLE

The state diagram in Figure 1(a) describes the classical seat belt controller of [9]. The controller waits for at most 32 seconds after the car key is turned on for the seat belt to be locked. If the seat belt was not locked, then the alarm will be on. Figure 1(b) shows the counter that implements the timer. Since it counts till 32, then at least 5 state bits are needed in the hardware implementation.

Suppose the car manufacturer decides to add a new safety feature and a designer implements it in a new component as shown in Figure 1(c). The new component monitors the same timer signal and forces the car to turn off if the seat belt was not locked before 6 seconds have elapsed. All we need to count for now is 6, and thus we only need 3 bits. However, we can see the potential for this reduction only when we co-optimize.

The same phenomenon is routinely faced in embedded system design. Examples include bus controllers with preemptive logic used in non-preemptive applications, cache IP that has pollution logic used in an application that never invalidates the data, an Internet Protocol processor that supports IPv6 used in a router that drops all IPv6 packets at the ingress, a microprocessor core that supports multiplication used in an application that is purely control dominated, etc.

## III. BASIC CONCEPTS

In this chapter we review the Metropolis development environment. Metropolis provides a framework that integrates a set of tools to operate and manipulate a common design representation. We then introduce and illustrate the SE technique with an example.

### A. Metropolis Overview

Metropolis provides a metamodel to support existing MoCs and accommodate new ones. Similar to other platforms like Polis [10], Ptolemy [11], SystemC [12], and SpecC [13]. Metropolis allows concurrent design of multiple processes and uses channels to communicate between them. It is different in that it introduces orthogonalities between (1.) computation versus communication, (2.) functionality versus architecture, and (3.) behavior versus performance. These processes are orthogonal since computation refinement is usually a manual process, functional and architectural specifications are decided by different groups, and performance constraints are specified by a group other than the behavior designers.

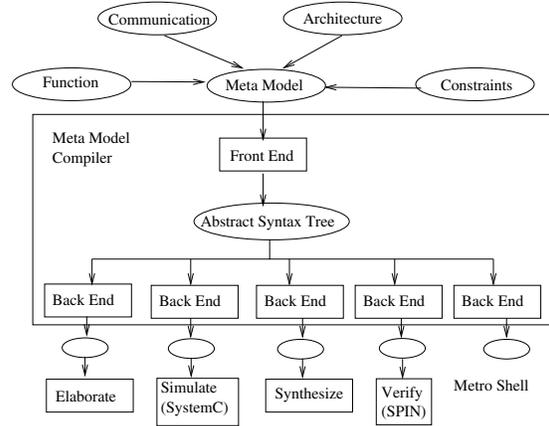


Fig. 2. Metropolis platform

The tools integrated in Metropolis allow designers to specify, partition, synthesize, and validate an embedded system. By partition we mean to decide which parts of the design get implemented by software that runs on programmable components and which parts get implemented directly in dedicated hardware devices. Synthesis compiles the software to optimized object code and hardware into optimized configurations of logic gate libraries. Validation checks whether the final implementation meets the properties in the specifications [14].

Metropolis uses MMM, an extension of the Java programming language with access restriction paradigms, as the design description language.

The diagram in Figure 2 shows the different components of the Metropolis platform. Metropolis provides several tools to manipulate the design. The two most commonly used are the Elaborator and the SystemC [12] code generator. The Elaborator parses the MMM specifications and generates an *abstract syntax tree* (AST). All the other tools in the Metropolis framework operate on the AST. The SystemC generator provides a path to simulation, verification, and synthesis. Metropolis also has another verification path to the SPIN [15] model checker.

**1) Design Methodology:** Metropolis allows a recursive design cycle with three phases. In Phase 1, the designer specifies the functional behavior of the design. This uses one or more MoC to describe (1.) the computation components, (2.) the communication components, (3.) the connections, and (4.) the hierarchy of the design. In MMM these are referred to as processes, media, interfaces and netlists respectively.

In Phase 2, the designer describes the target architecture platform of the design as a library of services. In Phase 3, the designer maps the functional specifications to the architecture services. In summary, the mapping is an intersection operation that constrains a functional interface to an architectural one.

The functional and architectural netlists may be developed by Metropolis designers and may also be provided as libraries. A platform vendor, for instance, may provide an architecture MMM of an FPGA to its customers. Similarly, a design team may develop a standard library of commonly used functional processes to promote reuse.

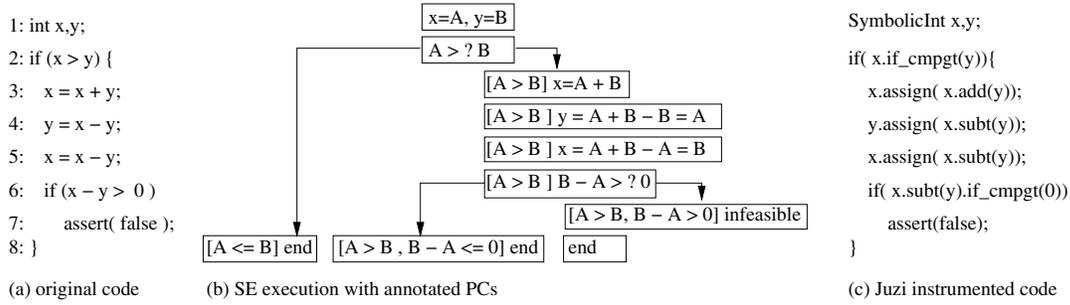


Fig. 3. Symbolic execution

2) **The Meta Model:** Metropolis uses a meta model to represent a design. Like the tagged-signal model [16], MMM is not tied to a particular MoC. MMM provides a set of computation and communication building blocks that can be customized to describe the semantics of many different MoCs thus enabling the modeling of heterogeneous systems.

MMM builds on the syntax and semantics of the Java programming language. It restricts itself to a subset of Java that includes inheritance, basic logical and arithmetic operators, loops, and conditionals. MMM introduces special class specifiers such as *process*, *media*, and *quantity* and supports method and field access control using modifiers like *eval*, *update*, and *port*. It also supports logical implication and equivalence operators in its embedded linear temporal logic (LTL) [17] used to specify constraints.

Processes are the active elements of the design and are typically used to perform computations via executing their thread methods. To decouple communication from computation, processes do not communicate directly with each other, they connect through interface ports and media. This allows the communication semantics to change without the need to change the processes.

Media are the communication components of the design and facilitate communication between processes and other media. They are passive, meaning they do not perform computation, but they may contain data and state.

The Metamodel supports two constraint formula types: LTL and logic of constraints (LOC). In addition, often LTL constraints form mappings expressing the equivalence between functional and architectural components.

A netlist connects the Metamodel components. Functional netlists usually consist of processes and media, while architectural netlists consist of processes, media, quantities, and constraints. Netlists may be connected hierarchically. A mapping netlist, for instance, can contain a functional netlist, an architectural netlist, and constraints between the two.

### B. Symbolic Execution

SE is a technique that explores all possible executions of a program by traversing all possible branching conditions while bookkeeping symbolic values for all variables involved in the program at each atomic step. For every branch, SE adds the condition for the branch it takes to a PC in conjunction. Upon completion, all PCs are evaluated. Some of the accumulated PCs may be proved unsatisfiable, and thus the code these

PCs annotate is proved dead since there is no setting for the program variables that satisfies its PC.

In Figure 3(b) we show the symbolic execution of the generic code in Figure 3(a). SE techniques start by initializing variables  $x$  and  $y$  to symbolic values  $A$  and  $B$ . At the first branch on line 2, the two choices are valid so they are both considered. The first,  $A \leq B$ , exits the block. The second,  $A > B$ , enters the body of the conditional statement. The values of  $x$  and  $y$  are updated at each line until the next branch is reached on line 6. At that point the first condition,  $B - A \leq 0$ , exits the block. The second,  $B - A > 0$ , leads to the assert statement. However the PC, accumulated from the branch on line 2, conflicts with the symbolic check on the values of  $x$  and  $y$ . Thus we can conclude that the assert statement is not accessible.

This shows how the SE technique can effectively detect dead code in control and data flow dominated circuits. With SE we can also annotate each line of code with all the PCs that allow the line to be executed. The tool Juzi [6] implements SE for Java code. It does so by instrumenting the Java-byte code of the Java class in question to use symbolic implementations for its variables. Figure 3(c) shows the Java code equivalent to the automatic Juzi instrumentation for the code in Figure 3(a). The variables  $x$  and  $y$  are declared as *SymbolicInt* variables and the arithmetic and Boolean operations and conditions applied to them are replaced by symbolic operations. The symbolic *assign*, *add*, and *subt* operations update the correspondence between  $x$  and  $y$  and the symbolic values  $A$  and  $B$  respectively. The *if\_cmpgt* updates and checks the PC with the different choices it makes. Juzi also introduces *lazy initialization* where it initializes a variable only when it is actually used rather than when it is declared. Juzi supports dynamically allocated structures and data structures with self-loops – e.g., linked lists, and respects method preconditions.

Under the hood, Juzi uses CVC-lite to decide a PC or to simplify it into an equivalent more compact PC. CVC-lite is a validity checker for typed first order logic with interpreted theories. It supports integer and real arithmetic, equality and inequalities, predicate subtypes, partial functions, uninterpreted functions, arrays, records, bitvectors, and quantifiers [8]. COSE leverages these capabilities to work on a high level of abstraction and detect invariants that are qualitatively different than those detected at the Boolean level.

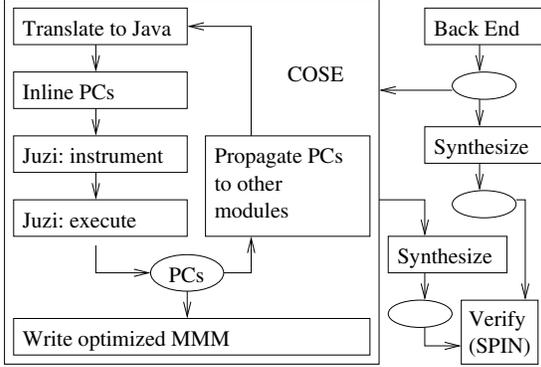


Fig. 4. COSE in Metropolis

#### IV. OUR APPROACH

Before we proceed to present the technical solution, we discuss two fundamental questions. First, one may question whether software can be developed before hardware is committed. Second, new versions of software may try to use hardware that was optimized away.

The software dependence on ready hardware question is methodology specific. The MMM assumes a complete design at every level of abstraction, and thus the reductions obtained by co-optimization apply only to that level. At any refinement, reductions need to be discarded and co-optimization can be applied again. The flexibility question is inherent in all models and is specific to the co-optimization technique. COSE provides the sets of constraints it detected and used to optimize other components. Thus it allows the designers to either stay within the framework of the optimizations or to discard the optimizations and run COSE again in case new conflicting constraints were found.

##### A. COSE Overview

Figure 4 shows how COSE is integrated into the Metropolis framework. COSE is an iterative process that uses SE and static analysis techniques to optimize a design at hand. It takes as input an MMM netlist  $\Phi$ , and a starting PC  $\sigma$  that could be empty. It selects one interesting process  $\Pi$  in  $\Phi$ , and it uses Juzi to symbolically execute the thread function in  $\Pi$ , with  $\sigma$  as the starting PC.

COSE uses Juzi and CVC-lite to perform SE. This is convenient since MMM is an extension to Java with access restrictions. COSE translates  $\Pi$  to Java and calls it  $\Pi_{\text{Java}}$ . The translation to Java is straightforward and works with inlining method calls and substituting ports by variables with Boolean validity guards to control read and write operations and allow a run-time protection mechanism. Then COSE compiles  $\Pi_{\text{Java}}$  into byte-code,  $\Pi_{BC}$ , and feeds that to the Juzi instrumentor class. COSE then symbolically executes the instrumented class and gathers the generated PCs.

COSE follows the usage of ports in the process and joins all the path conditions annotated to the code of the ports in disjunction into  $\sigma_i^{port}$ . Then each  $\sigma_i^{port}$  is treated as a constraint to the port and is propagated through the corresponding

connections in  $\Phi$ . COSE repeats these steps for every medium and process that connects to  $\Pi$  through a constrained port.

##### B. COSE Optimizations

We annotate each line of MMM code with the disjunction of the PCs generated for it from Juzi. We resolve and compact the PCs using CVC-lite. A PC that evaluates to false is *infeasible* and the code it annotates is dead-code and thus can be optimized away.

COSE can also perform range restriction optimizations. If, for example, COSE detects PCs restricting all indexes reading and writing into an array to a specific range, it can resize the array and use a smaller number of bits for the indexes.

COSE also checks for mutually exclusive blocks of code via evaluating the PCs of interesting blocks of code that use the same ports. Once two mutually exclusive blocks of code are detected, COSE adds a constraint to the netlist connecting the two blocks of code to declare they are mutually exclusive. Synthesis tools can use this to optimize communication between ports depending on the underlying MoC [18].

COSE generates optimized MMM modules that are annotated with detected and simplified PCs. In addition to conditional statements, COSE embeds the PCs it detected on ports as LTL constraints in the MMM module describing the netlist connections of the design. We pass the optimized and constrained MMM to a synthesis tool where we target reductions achieved by static analysis such as constant propagation, redundancy removal [19], [20], and observability don't care optimizations [21].

COSE operates only on the MMM level. It uses the netlist connecting the different components of a design only to follow the connections regardless of the semantics of the underlying MoC. Thus the optimizations and constraints computed by COSE are valid regardless of the underlying MoC.

##### C. Dependency Map and Constraint Propagation

COSE propagates constraints through ports. A port has an interface type and allows access to methods declared in the interface. More than one process may own ports with the same interface type, and media implement the interface methods. COSE follows the port declarations and usages in the entire netlist  $\Phi$  and forms a dependency graph. The nodes of the dependency graph are processes, media, and interfaces. The arcs are connected ports. All the connections in MMM are done with the `connect` keyword which facilitates building such a map. Connections in MMM have to be resolved statically and are done at the netlist level. This helps run SE on the component level only and helps COSE scale to large compositions. A PC is coded into a conditional constraint and is annotated either to the `thread` method of a process, or to the implementation of an interface method in a media.

##### D. Synthesis Reduction Techniques

COSE can use the native synthesis tools in Metropolis or can run public domain synthesis optimization tools like SIS [21] to optimize the components with the computed



2) **Results for Switch Fabric:** We assumed the deployment of the switch fabric design in Figure 5 in multiple scenarios. The first scenario is with network elements that produce IP only traffic. The second scenario was with local network elements that are limited to the use of few output ports.

We modeled the IP-only traffic by a CreateInputs process. The following fragment of pseudo code from CreateInputs shows how the protocol field is set to IP\_PROTOCOL for valid inputs.

```
int inIdx;
for ( inIdx = 0 ; inIdx < NUM_IN; inIdx++ ) {
  inBuff[inIdx].valid = chooseBoolean();

  if (inBuff[inIdx].valid == 1) {
    // if (chooseBoolean() == 1)
    inBuff[inIdx].pkt.protocol = IP_PROTOCOL;
  } else
  // inBuff[inIdx].pkt.protocol = ATM_PROTOCOL;
  ... }
}
```

Similar to Figure 3, COSE instruments all fields of the pkt as SymbolicInt and thus detects the path condition  $PC_{prtcl}$ ,  $\text{inBuff[inIdx].protocol} == \text{IP\_PROTOCOL}$ , in all execution branches of the CreateInputs process. COSE propagates  $PC_{prtcl}$  to SFComputeSchedule through the InputIfc ports. COSE was able to determine that all blocks of code conditioned with  $\text{if}(\text{inBuff[inIdx].protocol} == \text{ATM\_PROTOCOL})$  are infeasible in all branches of execution of SFComputeSchedule. COSE annotates all these blocks of code with an  $\text{if}(\text{false})$  PC and thus optimizes them out. In addition, COSE annotated the PacketIfc and the ScheduleIfc interfaces with  $PC_{prtcl}$  since it still holds across all execution branches in SFComputeSchedule. Then in turn, COSE propagated the  $PC_{prtcl}$  to the SFPerformTransferProc process. All blocks of code therein conditioned with  $\text{if}(\text{inBuff[inIdx].protocol} == \text{ATM\_PROTOCOL})$  were optimized out.

With 4 input ports, 4 output ports, and an  $8 \times 16$  internal packet buffer, it took COSE 3 hours and 12 minutes to complete this optimization with 36,816 symbolic integer variables and more than 2 million branches of execution.

We modeled the network with limited output port accessibility with a CreateInputs process that sends control packets with a limited destination port value.

```
int inIdx;
for (inIdx = 0; i < NUM_IN; i++) {
  inBuff[inIdx].valid = chooseBoolean();
  if (inBuff[inIdx].valid == 1) {
    int choiceDataVsControl = chooseBoolean();
    if (choiceDataVsControl == 1) {
      inBuff[inIdx].pkt.data[0] = CTRL_PKT;
      inBuff[inIdx].pkt.data[1] =
        makeDestIPAddress();
      inBuff[inIdx].pkt.data[2] =
        chooseInt().remainder(4);
      ...
    } else { // setup data packet
      ... }
    }
}
```

COSE was able to detect the path condition  $PC_{port}$ ,  $(\text{inBuff[inIdx].pkt.data[0]} == \text{CTRL\_PKT}) \text{ AND } (\text{inBuff[inIdx].pkt.data[2]} \leq 4)$ . COSE propagated the  $PC_{port}$  condition, through the InputIfc interface, to SFUpdateTablesProc and then annotated all the LookupTableIfc interfaces with  $\text{lookupTable[tableIdx].outputPort} \leq 4$ . COSE propagated the latter PC to SFPerformTransferProc and annotated the only block of code that writes to the output ports with it.

```
if (outPortIdx < 4) { // COSE ANNOTATION
```

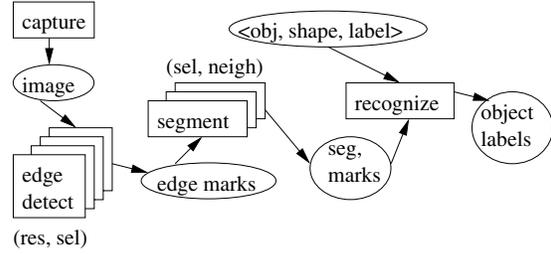


Fig. 7. Flow diagram for the ObjectID application

```
outBuff.[outPortIdx].valid = 1;
outBuff.[outPortIdx].pkt = getPcktFromScheduleAndMemory();
... } // END COSE ANNOTATION
```

With this annotation COSE enabled the synthesizer to drop the 4 output ports that are never written to or validated. COSE ran with 61,018 symbolic integer variables and took 4 hours and 7 minutes to inspect all feasible branches of execution and compute this optimization.

### B. Case Study: Video Surveillance

Figure 7 shows an image processing video surveillance application, also called object identification, that was developed in an MS project [25]. The system accepts an image and produces labels of the objects with their coordinates in the image. One or more edge detectors scan the image and produce edge marks that are visited by one or more segmentation algorithms to produce segmentation marks. The segmentation marks partition the image with the borders of the main shapes in it. Then a pattern recognition algorithm is used to match the shapes in the segmented image to a library of objects. The application instantiates 4 edge detectors and 3 segmentors and uses them depending on the resolution of the requested edge and recognition. We assume the system was designed for military usages that need high resolution analysis, and then it is used for home surveillance purposes where lower resolutions are suitable. If only home surveillance resolution is desired, just two detectors and one segmentor are needed.

1) **MMM Model for ObjectID:** We implemented the object identification example in MMM. The system originally took 16 weeks of work by one designer and is around 4000 lines of C and RTL code. We modeled it with around 1255 lines of MMM code. The diagram in Figure 8(a) shows the high abstract view of the ObjectID application, and the diagram in Figure 8(b) shows the detailed class diagram.

The OIDImage media implements OIDImageObjectIfc which defines interfaces for object identification and inherits image, edge, and segment interfaces. An OIEdge process accesses the OIDImage media through owning a port of type ImageEdgeIfc. OIEdgeGaussianEdge extends the OIEdge process and implements one type of edge detectors which uses a Gaussian mask to differentiate pixels. We only named one type of edge detectors for brevity. Other well known techniques for edge detection such as the Sobel, Prewitt, Laplacian of Gaussian (LoG), and the Canny detectors [26] may be used. Similarly to OIEdge, OIEdgeSegment owns a port of type

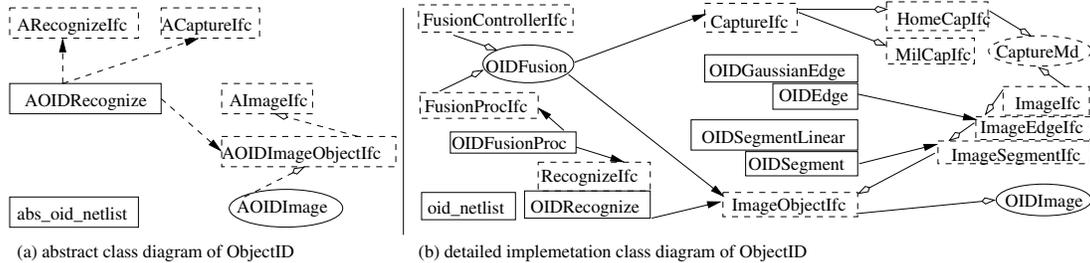


Fig. 8. Levels of abstraction of the Object ID application

ImageSegmentIfc, and OIDSegmentLinear implements one type of segmentation algorithms. The OIDFusion medium owns a CaptureIfc, and an ImageObjectIfc. It also implements the FusionControllerIfc and FusionProcIfc interfaces. OIDFusionProc owns FusionProcIfc and RecognizeIfc interfaces. The OIDRecognize process also owns similar interfaces.

The netlist `oid_netlist` instantiates 4 `OIDGaussianEdge` processes, 2 `OIDSegmentLinear` processes, one `OIDImage` medium, one `OIDFusion` medium, and one `OIDFusionProc` process. It also connects them appropriately through their owned and implemented ports. `CaptureIfc` and the object part of `ImageObjectIfc` are open to the outside world where the system can accept controlled input.

2) **ObjectID Dependency Map:** Figure 9 shows the code of the ObjectID netlist, `OIDFusion`, and `OIDGaussianEdge`. The arrow on the connect method call in the netlist constructor links the usage of the `oid_fusion_ifc` port to its implementation in `OIDFusion`. If the capture and the recognition interfaces happen to be set to 120 and 105 then the body of the thread method of the `OIDGaussianEdge` detectors instantiated as `edge3` and `edge4` will be dead code since the first statement in the condition is going to be always false. The arrows in the figure show how the dependency map was built from instantiations and connections in the netlist and usages of ports in process classes.

3) **COSE ObjectID Results:** In the ObjectID, we assumed a capture medium implementation that requires a low resolution. We were able to detect infeasible PCs on lines that activate the two high resolution edge detectors in one COSE iteration. In the second iteration and after propagating the infeasible PCs we were also able to detect dead code for the lines that activate one of the segmentation processes. We were able to reduce the system by 2 edge detector components and one segmentation component. It took COSE 15 minutes to instrument and symbolically execute the ObjectID code.

## VI. RELATED WORK

Compared to sequential optimization techniques from logic synthesis, specifically, those based on sequential don't cares [27], COSE operates at much higher level of abstraction than Boolean netlists—we treat high-level constructs such as integer variables, memories, etc., natively. We also make use of the decomposition of the design into modules. This leads

to more compact models, reducing run-time complexity which has been a major drawback to sequential logic synthesis.

Analyzing software components with Boolean methods is far beyond the reach of current tools—there is too much state associated with the instruction set. We have in the past experimented with verifying a switch fabric with VIS [28]. We implemented the scheduling in hardware, and it was considerably simpler than the scheduler employed in this paper. However, verification still suffered from BDD-blowup, and we were unable to verify elementary properties of a 2-input 2-output shared memory switch with 4 packets of buffering, no control packets, and no support for mixed-mode traffic.

Our SE technique is different from symbolic reachability analysis [28] in many aspects. First, it does not examine the transition system of a component in whole and rather looks at the transition of interleaving variables that define the state space via executing the code line by line. Also it only considers reachable states via pruning infeasible PCs.

Prior dead-code detection techniques [29], [30] make use of conservative static analysis and can therefore erroneously classify paths as alive; in contrast, SE coupled with a decision procedure gives more accurate results for programs with bounded loops and recursion.

Similar to VeriSoft [31], COSE performs a *state-less* search: it systematically explores an *execution tree*, which represents all execution paths up to a desired bound, and backtracks using re-execution, ensuring that all (feasible) paths are explored. This contrasts with the *state-full* search of SPIN [15], which stores states using hashing. COSEs use of symbolic execution also contrasts with VeriSoft and SPIN, who do not use off-the-shelf decision procedures and require the user to a priori provide (tight) bounds on integer inputs.

## VII. SUMMARY

We introduced the use of SE to co-optimize embedded systems, presented COSE and implemented a prototype that uses Java inlining. We integrated COSE in the Metropolis framework and showed it can lead to significant optimizations on representative examples. COSE is a co-optimization mechanism that can find constraints in software and effectively propagate them to hardware. These constraints may not be observable by designers and lead to optimizations that are qualitatively different than reductions possible with localized compiler optimizations and Boolean level transformations.

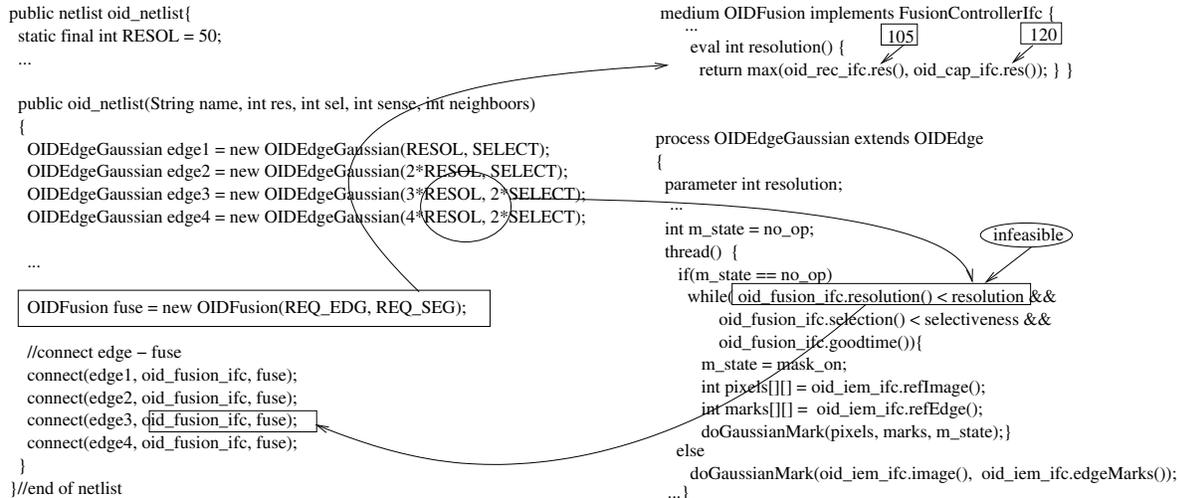


Fig. 9. Dependency map of ObjectID

Given the different levels of abstractions possible in Metropolis, COSE can accompany the concretization process one step at a time. COSE guarantees that the result of its optimization is functionally equivalent to the original design. In the future, we plan to optimize COSE via integrating it with a native difference equation solver instead of calling CVC-Lite. We are also exploring the use of SE to optimize linking programs across compilable software modules.

## REFERENCES

- [1] F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe, "Modeling and designing heterogeneous systems," tech. rep., Cadence Berkeley Laboratories, Nov 2001.
- [2] F. Balarin et al., "Metropolis: an integrated electronic system design environment," in *Computer* 36, Apr 2003.
- [3] M. Chiodo, "Optimization and synthesis for complex reactive embedded systems by incremental collapsing," in *International symposium on Hardware/software codesign*, 2002.
- [4] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Design Automation Conference*, 1995.
- [5] M. Chiodo, P. Guisto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli, "Synthesis of mixed software-hardware implementation from CFSM specifications," in *International Workshop on Hardware-Software Codesign*, 1993.
- [6] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr 2003.
- [7] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.
- [8] C. Barrett and S. Berezin, "Cvc lite: A new implementation of the cooperating validity checker," in *Computer Aided Verification*, April 2004.
- [9] P. S. Roop, A. Sowmya, and S. Ramesh, "k-time forced simulation: A formal verification technique for ip reuse," *International Conference on Computer Design*, 2002.
- [10] F. Balarin et al., *Hardware-software codesign of embedded systems: The POLIS approach*. Academic Publishers, Boston, 1997.
- [11] E. A. Lee, "Overview of the ptolemy project," tech. rep., University of California, Berkeley, Jul 2003.
- [12] T. Grotker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [13] M. Fujita and H. Nakamura, "The standard specc language," in *International Symposium on Systems Synthesis*, 2001.
- [14] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification," in *Formal Methods and Models for Co-Design*, July 2006.
- [15] G. Holzmann, "The model checker SPIN," in *IEEE Transactions on Software Engineering*, May 1997.
- [16] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," *Proc. of the IEEE*, vol. 85, 1997.
- [17] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *International Symposium on Protocol Specification, Testing and Verification*, 1996.
- [18] O. Bringmann, W. Rosenstiel, and A. Siebenborn, "Conflict analysis in multiprocess synthesis for optimized system integration," in *International conference on Hardware/software codesign and system synthesis*, 2005.
- [19] H. Mony et al., "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design*, Nov. 04.
- [20] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Formal Methods in Computer-Aided Design*, November 2000.
- [21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," tech. rep., Univ. of California, Berkeley, May 1992.
- [22] J. Turner and N. Yamanaka, "Architectural Choices in Large Scale ATM Switches," *IEICE Transactions*, 1998.
- [23] A. Prakash, A. Aziz, and V. Ramachandran, "Randomized Parallel Schedulers for Switch-Memory-Switch Routers: Analysis and Numerical Studies," in *IEEE Infocom*, 2004.
- [24] J. Networks, "High speed switching device," US Patent 5,905,726, 1999.
- [25] F. Zaraket, "Optimal Fusion and Objective Comparison of Edge Detectors," Master's thesis, American University of Beirut, 2001.
- [26] D. Marr and E. Hildreth, "Theory of edge detection," in *the Royal Society of London*, 1980.
- [27] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential Synthesis Using SLS," *IEEE Trans. Computer-Aided Design*, vol. 19, Oct. 2000.
- [28] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS," Nov. 1996.
- [29] R. Bodek and R. Gupta, "Partial dead code elimination using slicing transformations," in *ACM Conference on Programming language design and implementation*, 1997.
- [30] Y. A. Liu and S. D. Stoller, "Eliminating dead code on recursive data," in *Static Analysis Symposium*, 1999.
- [31] S. Chandra, P. Godefroid, and C. Palm, "Software model checking in practice: an industrial case study," in *International Conference on Software Engineering*, 2002.

# Cross-Entropy Based Testing

Hana Chockler\*, Eitan Farchi\*, Benny Godlin\* and Sergey Novikov\*

\*IBM Haifa Research Laboratory

Mount Carmel, Haifa 31905, Israel

Email: {hanac,farchi,godlin,novikov}@il.ibm.com

**Abstract**—In simulation-based verification, we check the correctness of a given program by executing it on some input vectors. Even for medium-size programs, exhaustive testing is impossible. Thus, many errors are left undetected. The problem of increasing the exhaustiveness of testing and decreasing the number of undetected errors is the main problem of software testing. In this paper, we present a novel approach to software testing, which allows us to dramatically raise the probability of catching rare errors in large programs. Our approach is based on the *cross-entropy method*. We define a performance function, which is higher in the neighborhood of an error or a pattern we are looking for. Then, the program is executed many times, choosing input vectors from some random distribution. The starting distribution is usually uniform, and it is changed at each iteration based on the vectors with highest value of the performance function in the previous iteration. The cross-entropy method was shown to be very efficient in estimating the probabilities of rare events and in searching for solutions for hard optimization problems. Our experiments show that the cross-entropy method is also very efficient in locating rare bugs and patterns in large programs. We show the experimental results of our cross-entropy based testing tool and compare them to the performance of ConTest and of Java scheduler.

## I. INTRODUCTION

*Software testing* (also called *simulation-based verification*) is a family of analyses that involve an automatic or semi-automatic exploration of the state space of a program. Simulation-based verification is traditionally used in order to check the program with respect to some input vectors [2]. It is widely used today as a primary means for checking the correctness of programs. Thus, in simulation-based verification, the challenge of making the verification process as exhaustive as possible is crucial. Each input vector induces a different execution of the program, and a program is correct if it behaves as required for all possible input vectors. In the ideal world, a program would be tested on all input vectors. This approach, however, is infeasible even for medium-size programs, whereas today it is common to find programs with a few million lines of code. To make matters worse, reuse of components (that may have never been tested with the new use in mind) makes it possible to assemble products orders of magnitude larger, in the same time.

Since simulation-based verification is a technique that replaces the infeasible task of checking all input vectors, it is very important to increase the capability of simulation-based verification to find errors and to make the testing process as thorough as possible. The research in this area focuses on several different directions, which we describe in more detail in Section II. While the existing techniques have been

successful in specific applications, none of them has proven successful in efficiently finding a large variety of rare bugs in large programs. The problem is especially acute in concurrent programs, which many have exponentially many possible behaviors resulting from different possible schedulings of threads.

In this paper we propose a new approach to testing of large programs, which is based on the *cross-entropy method*. The cross-entropy (CE) method is a generic approach to rare event simulation [19]. It derives its name from the cross-entropy or the Kullback-Leibler distance, which is a fundamental concept of modern information theory [13]. It is an iterative approach, and is based on minimizing the cross-entropy or the Kullback-Leibler distance between probability distributions. The CE method was motivated by an adaptive algorithm for estimating probabilities of rare events in complex stochastic networks [17]. Then, it was realized that a simple modification allows to use this method also for solving hard combinatorial optimization problems, in which there is a performance function associated with the inputs. The CE method involves an iterative procedure, where each iteration consists of two phases:

- 1) Generate a random data sample according to a specified mechanism.
- 2) Update the parameters of the random mechanism based on the data to produce a “better” sample in the next iteration, where “better” is chosen according to the predefined performance function.

A sample is evaluated according to a predefined performance function. The procedure terminates when the “best” sample, that is, a sample with the maximal value of the performance function (or, if the global maximum is unknown in advance, with a sufficiently small relative deviation), is generated. The CE method is used in many areas, including buffer allocation [1], neural computation [8], DNA sequence alignment [12], scheduling [14], and graph problems [18].

In testing, in order to make the CE method applicable we shift the focus from searching for bugs to focusing our attention on the most error-prone areas of a program. The following example helps to clarify the difference between the approaches. Assume that we are given a program, which we want to test for buffer overflow errors. A bug-oriented testing tool searches for executions in which the buffer is overflowed. These executions might be very rare and might escape our testing efforts. In cross-entropy-based testing, we

direct the execution to the areas in which the buffer reaches its maximal capacity. The advantage of this approach is that this area, while can be small, is not of negligible size, and can be discovered during random testing. Also, if there exists an erroneous execution in which the buffer is overflowed, it surely occurs in this area.

We argue that many common bugs and patterns which may contain bugs have a natural performance function. For example, in case of the buffer overflow, the natural performance function for an execution gives the value which is equal to the maximal size of the buffer in this execution. We describe many more examples of common bugs and bug patterns and their associated performance functions.

Our way of using CE method in testing is suitable for programs with many points of non-deterministic decisions. Informally, such programs create a large control-flow graph with many branching points, which allows us to view the testing setting as a variation of a graph optimization problem. While a serialized program can, in theory, have many points with non-deterministic decisions (for example, statements conditional on the result of coin-tossing), the most common example of such programs is concurrent programs. In concurrent programs, at each synchronization point the decision of which process (or thread) makes the next step is made by the scheduler and can be viewed as non-deterministic when analysing the program. In this work, we focus on concurrent programs and apply the CE method to errors related to concurrency (such as a way of scheduling the threads that leads to buffer overflow). Our work can, in theory, be extended to arbitrary non-deterministic programs. However, we believe that testing concurrent programs is the area in which CE method has the maximal advantage.

We implemented our algorithm in a tool named *ConCenter*, which stands for “Concurrent Cross-Entropy based Error Revealer”. We provide the details of implementation and the experimental results of running ConCenter on several programs with different bug patterns<sup>1</sup>. We compare the performance of ConCenter with the performance of ConTest, a randomized testing tool for concurrent programs developed in IBM [9]. We show that ConCenter performs better than ConTest by an order of magnitude.

## II. RELATED WORK

In this section, we briefly survey the existing work in the area of improving the capability of simulation-based verification to find bugs.

One direction is to focus on bugs that manifest themselves under a heavy load on the system during runtime, such as buffer overflows, timeouts, etc. The traditional method of testing for these bugs is called stress-testing. The method takes single-thread tests and creates stress tests by cloning them many times and executing them simultaneously (see, for example, [10]).

<sup>1</sup>The executable of ConCenter with some test programs is available from the authors on request.

Another approach to testing is to create random interference in the scheduler which leads to randomized interleavings of the thread executions (these tools are also called “noise makers”) [9], [22]. The interference is created by injecting noise to the scheduler forcing the scheduler to create uncommon interleavings, which are rare during usual execution. However, a chance of finding a very rare bug is small, since the random distribution (created by injecting noise) is not adjusted to a specific pattern.

The most commonly used method to maximize the exhaustiveness of testing is *coverage estimation*. There is an extensive research simulation-based verification community on coverage metrics, which provide a measure of exhaustiveness of a test [23]. Coverage metrics are used in order to monitor progress of the verification process, estimate whether more input sequences are needed, and direct simulation towards unexplored areas of the program. Essentially, the metrics measure the part of the design that has been activated by the input sequences. For example, code-based coverage metrics measure the number of code lines executed during the simulation. There is a variety of metrics used in coverage estimation (see, for example, [7], [15], [16], [23], [26]), and they play an important role in the design validation effort [24]. Still, since there is no special effort directed to a possibly erroneous pattern in the code, rare bugs can remain in the code even after testing reached a high coverage of it.

Existing testing methods that look for predefined patterns in programs rely on heuristics specifically targeted to a pattern, for example, delaying read/write instructions [3], or denying the application certain services [25]. This approach is quite efficient for finding predefined patterns, however, it requires a different heuristic to be invented for each new bug.

An approach based on exploiting genetic algorithms [11] in testing led to several prototypes of testing tools implementing this idea (see, for instance, [20], [21]). These algorithms, however, normally require a very long execution time, and they have also been shown to be ineffective in finding rare and hard-to-reproduce bugs.

## III. PRELIMINARIES

### A. The Cross-entropy Method in Optimization Problems

The cross-entropy (CE) method was developed in order to efficiently estimate probabilities of rare events, where an event  $e$  is considered a rare event if its probability is very small, say, smaller than  $10^{-5}$ . In this method, we are given a very large probability space and a function  $S$  from this space to  $\mathbb{R}^+$ , and we say that  $e$  occurs on an input  $X$  (from the probability space) if  $S(X) > \gamma$  for some predefined value  $\gamma \in \mathbb{R}^+$ . Since the space is very large, it is infeasible to search it exhaustively, therefore the estimation of the probability  $l$  of  $e$  is made by sampling. A straightforward way to estimate  $l$  is to draw a random sample according to the given probability distribution  $f$  on inputs, and then estimate  $l$  by examining the sample. The problem is, that when  $e$  is a rare event, the sample might have to be very large in order to estimate  $l$  accurately. A better way is to draw the sample according to some other

probability distribution  $g$  that raises the probability of  $e$ . The ideal probability distribution here would be  $g_l$ , which gives the probability 0 to inputs that do not contain  $e$ . The CE method attempts to approximate  $g_l$ . The distance between two distributions that is used in this approximation is the Kullback-Leibler “distance” (also called cross-entropy). The Kullback-Leibler “distance” between  $g$  and  $h$  defined as:

$$\mathcal{D}(g, h) = E_g \ln \frac{g(X)}{h(X)} = \int g(x) \ln g(x) dx - \int g(x) \ln h(x) dx.$$

Note that this is not a distance in the formal sense, since in general it is not symmetric. Since  $g_l$  is unknown, the approximation is done iteratively, where in iteration  $i$  we draw a random sample according to the current probability distribution  $f_i$  and compute the (approximate) cross-entropy between the  $f_i$  and  $g_l$  based on this sample. Then we construct  $f_{i+1}$  by updating  $f_i$  based on the cross-entropy result. The reader is referred to Appendix for more formal explanation and to the book on cross-entropy for the complete description of the method [19].

Our method of testing large program is based on the application of CE method to graph optimization problems. In these problems, we are given a (possibly weighted) graph  $G = \langle V, E \rangle$ , and the probability space is the set of paths in  $G$  represented by the sets of traversed vertices. This setting matches, for example, the definitions of the traveling salesman problem and the Hamiltonian path problem in the context of CE method. This is also the setting which we use in our approach. Our goal is to find executions of the program under test in which the system enters a predefined state. For example, in order to test stack overflows, we are interested in executions in which the stack occupancy is maximal. As we explain in Section III-B, we represent the program under test as a graph. Then, paths in this graph correspond to executions of the program. We define  $S$  (which we call a *performance function*) so that  $S(X)$  reaches its global maximum on paths in which the system is brought to the state we are testing. As we argue in Section IV-C, many common bugs and bug patterns admit a natural performance function. The probabilities can be assigned to edges or to vertices of the graph (depending on how the sample is drawn). W.l.o.g, we assume that the probabilities are assigned to edges. We start with uniform probability distribution. In each iteration  $i$ , we sort the sample  $\mathcal{X}_i = \{X_1, \dots, X_N\}$  generated in this iteration in ascending order of their performance function values. That is,  $S(X_1) \leq S(X_2) \leq \dots \leq S(X_N)$ . For some  $q \ll 1$ , let

$$Q(\mathcal{X}_i) = \{X_{[(1-q)N]}, X_{[(1-q)N+1]}, \dots, X_N\}$$

be the best  $q$ -part of the sample. The probability update formula in our setting is

$$f'(e) = \frac{|Q(e)|}{|Q(v)|}, \quad (1)$$

where  $e \in E$  is an edge of  $G$  that originates in the vertex  $v$ ,  $Q(v)$  are the paths in  $Q(\mathcal{X}_i)$  which go through  $v$  and  $Q(e)$  are the paths in  $Q(\mathcal{X}_i)$  which go through  $e$ . Intuitively, the

edge  $e$  “competes” with other edges that originate in  $v$  and participate in paths in  $Q(v)$ . We continue in the next iteration with the updated probability distribution  $f'$ . The procedure terminates when a sample has a relative standard deviation below a predefined threshold parameter (usually between 1% and 5%).

*Remark 3.1 (Smoothed updating):* In optimization problems involving discrete random variables, such as graph optimization problems, the following equation is used in updating the probability function instead of Equation 1:

$$f''(e) = \alpha f'(e) + (1 - \alpha) f(e), \quad (2)$$

where  $0 < \alpha \leq 1$  is the *smoothing parameter*. Clearly, for  $\alpha = 1$  we have the original updating equation. Usually, a value of  $\alpha$  between 0.4 and 0.9 is used. The main reason why the smoothed updating performs better is that it prevents losing a good sample forever (if one of its edges is assigned 0 in one of the iterations).

## B. Definitions

In this work we focus on finite multi-threaded programs. Let  $t$  stand for the number of threads in the program. As all executions of the program are finite, we can talk about the unwound code of the program. That is, the code of each loop is duplicated the maximum number of times it may run and all function called are embedded in the code of the main function.

*Definition 3.2 (PL):* *Program location* (PL) is a line number in unwound code of the program.

*Definition 3.3 (CFG<sub>i</sub>):* A *control flow graph* (CFG <sub>$i$</sub> ) of thread  $i$  ( $i \in [t]$ ) is a directed graph  $G_i = \langle L_i, E_i, \mu_i \rangle$  where  $L_i$  is the set of all program locations in the unwound code of the thread,  $E_i$  is the set of edges such that  $\langle v, u \rangle \in E_i$  if a statement at location  $u$  can be executed immediately after the statement at location  $v$ , and  $\mu_i \in L$  is the initial program location of the thread.

*Definition 3.4 (PLV):* *Program location vector* (PLV)  $\vec{v}$  is a  $t$  dimensional vector such that for each  $i \in [t]$   $v_i \in L_i$ .

At each moment  $m$  during the execution of the program, we say that the execution is at PLV  $\vec{v}$  if for each  $i \in [t]$   $v_i$  is the next program location to be executed in thread  $i$ .

Clearly, the set of all PLVs is equal to the cross product of the  $L_i$ s.

*Definition 3.5 (JCG):* The joint control graph (JCG) of the tested program is a graph  $\langle V, E \rangle$  whose vertices are the PLVs. There is an edge in the JCG between vertices  $u$  and  $w$  if there exists an execution path in which  $w$  is the immediate successor of  $u$ .

Note that at each moment only one thread can make a move. Therefore, the branching degree of each vertex is at most  $t$ . Since the code is unwound, every statement in it is executed at most once and the statements are executed in the increasing order of their program locations. Therefore, JCG is a finite directed acyclic graph (DAG). The source vertex of the JCG is a PLV which is composed of the initial PL  $\mu_i$  for each thread  $i$ .

*Definition 3.6 (PF):* Probability function  $PF : V(JCG) \times [t] \mapsto [0, 1]$  such that the probability sum over the outgoing edges of each vertex is 1.

This function defines for each vertex  $v$  and each of its outgoing edges  $e_i$  the probability of the thread  $i$  to advance when the execution reaches  $v$ . If not all threads are enabled at  $v$ , we take the relative probabilities of the enabled ones. If  $T_{en} \subseteq [t]$  is the set of the enabled threads at this moment then the relative probabilities are:

$$RP(v, i) \doteq \begin{cases} \frac{PF(v, i)}{\sum_{j \in T_{en}} PF(v, j)} & \text{if } i \in T_{en} \\ 0 & \text{otherwise} \end{cases}$$

*Definition 3.7 (Visible thread state):* A visible thread state (VTS)  $s$  of a thread  $i$  is the pair  $\langle v, \sigma, \tau \rangle$  where  $v \in L_i$  denotes the current PL,  $\sigma$  denotes the valuation of all local variables, and  $\tau$  denotes the valuation of the global variables accessed by the statement at program location  $v$ .

Intuitively, the set of visible thread states are all possible states of the system as visible by the thread when it accesses various parts of this system.

We assume that correct locking policy was implemented in the program (this can be checked statically). Therefore, any VTS that can be reached by any program execution, can be also reached by some program execution when context switches are allowed only at locking or unlocking statements. Thus, under such context-switch policy the only vertices of the JCG which may have more than one successor are those PLVs with program locations of lock or unlock statements. Then, all vertices with a single successor can be collapsed without losing any synchronization information, and therefore we can assume that the JCG graph contains only program locations of locking/unlocking statements<sup>2</sup>. This restriction on the context-switch policy was introduced in [4], [5].

For a single execution of the program, we call the sequence of vertices of JCG that it visits an *execution path* in JCG.

#### IV. CROSS-ENTROPY FOR TESTING

##### A. Algorithm

The algorithm starts with choosing the performance function  $S$  according to the state of the system to which we want to converge (we list some of the more natural performance functions in Section IV-C). A concurrent program is viewed as a JCG, which is a product of DAGs of its threads. Each DAG corresponds to the unwound program of a thread. The JCG is not constructed in advance - edges between PLVs are discovered during the execution of the algorithm.

Essentially, the algorithm simulates a random walk on the control graph of the program by deciding, at each synchronization point, which thread makes the next step<sup>3</sup>. The structure of the algorithm is presented in Figure 1, and we also describe it in more detail below.

- 1) The probability distribution table contains probability distribution on edges and is updated at each iteration.

<sup>2</sup>We assume a single statement per program location.

<sup>3</sup>Recall that we assume that all nodes are synchronization points.

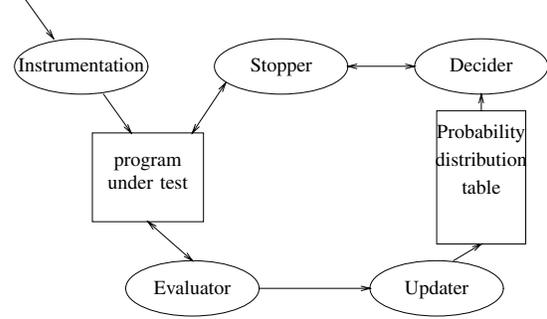


Fig. 1. The structure of our algorithm

Initially, the edges are unknown, so the table is empty, assuming uniform distribution.

- 2) The program is instrumented by adding callbacks at synchronization points. This enables the algorithm to stop the execution at these points and decide which edge is going to be traversed next (that is, which thread makes a move). At this point, the algorithm gathers the knowledge of the edges of JCG and stores it.
- 3) At each iteration, the algorithm performs the following tasks:
  - a) The instrumented program is executed a number of times that is sufficient to collect a meaningful sample (the number of executions depends on the size of JCG). Executions are forced to perform scheduling decisions according to the current probability distribution on edges. Stopping the execution and deciding which thread is allowed to run next is performed by separate components in our implementation.
  - b) The executions are used in order to compute the new probability distribution (see Equation 1 and Equation 2). We discuss the choice of the parameters  $q$  and  $\alpha$  in Section V.
- 4) The algorithm terminates when the collected sample of the current iteration has a sufficiently small relative standard deviation (between 1% and 5%, as discussed in Section III-A).

##### B. Heuristics for improving performance

*Dealing with unwinding:* To generate the full JCG of the unwound program (without actually generating the unwound code), a program location is composed from the line number in the original (not unwound) code and an additional parameter, which reflects the unwinding. This parameter is a vector of values of loop counters for all loops the program is currently in. The example in Figure 4.1 illustrates why a single counter is not enough to determine the program location uniquely.

*Example 4.1:* The program block in Figure 2 contains nested loops. The values of  $maxI$  and  $maxJ$  are received from other thread and define the number of iterations of each loop. Consider two cases: (1)  $maxI = 1$  and  $maxJ = 2$ ,

```

assert(maxI ≤ 10);
assert(maxJ ≤ 10);
for(i = 0; i ≤ maxI; i++)
    for(j = 0; j ≤ maxJ; j++)
        foo(i, j);

```

Fig. 2. An example of nested loops

and (2)  $maxI = 2$  and  $maxJ = 1$ . Consider the second call to  $foo(i, j)$ . In each of these cases, this call will appear in a different place in the unwound program, but the value of a single counter in both cases would be the same.

For the general case, we present a better parameter, using which, a 1-1 mapping can be created for any inner iterative structure of the program. This parameter will take the form of a stack. Each time the thread execution reaches a loop it pushes a new counter with value 0 on top of the stack, before entering the loop. Each time the execution starts a new iteration of a loop it increases the value of the top counter on the stack. Each time the execution exits the loop scope we pop the top counter of the stack. The same idea can be applied to function calls. Currently we do not test programs with recursion, though the method can be applied to recursive programs after minor changes.

*Modulo reduction:* In practice, the JCG of the unwound program can be huge even for medium-size programs. The problem with this is twofold. First, the graph itself may be too big to fit in the memory and to search efficiently. Second and more important, the set of execution samples generated at each iteration of the algorithm may be too sparse when projected on the JCG. In such case, the probability is updated only for a small fraction of the nodes. The consequence is that the algorithm converges to an arbitrary local maximum, instead of the target global maximum. We solve both problems by introducing the *modulo reduction*, in which the counters in program locations are computed modulo some small integer. The modulo reduction creates a *modulo joint control graph* (MJCG) from the original JCG. Each vertex of JCG is mapped to a vertex of MJCG by taking the modulo value of all its counters. The modulo reduction decreases the size of the graph significantly, however, it does not preserve the desired 1-1 mapping to unwound program locations. Our experiments show that with some fine tuning, it is possible to find the optimal modulo parameter, in which the many-to-one mapping does not confuse the CE computation, and yet, the MJCG is dense enough to allow the CE method to converge to a global (and not local) maximum.

### C. Applications

In this section we show that CE method is useful for testing. The necessary prerequisite of using the CE method to find a rare pattern or bug is the ability to define a performance function for this pattern. We show that many (if not the majority) of potential problems in programs admit a natural performance function, and thus can be discovered using the CE method. As an easy example, consider searching for

buffer overflows. Using the CE method, we can direct the executions of the program to areas where the buffer occupancy is maximized. The natural metrics for this problem is the number of elements in the buffer. The following is a partial list of other common patterns in programs, often associated with bugs, for which there exist natural metrics, and which, therefore, can be found using the CE method.

- **Discovering data races:** the function is the number of shared resources accessed during the execution.
- **Testing error paths:** the function is the number of error paths taken during the execution.
- **Incorrect use of synchronization primitives:** the function is the number of calls to mutually exclusive functions in an execution.
- **Bugs caused or related to wait-notify situations:** the function is the number of lost notifies.
- **Testing recovery from multiple failures of threads/tasks:** the function is the number of canceled threads/tasks.
- **Simulation of the environment that triggers many exceptions:** the function is the number of generated exceptions.
- **Not releasing resources properly or exhaustion of resource pools:** the function is the number of allocated resources minus the number of released ones.

## V. IMPLEMENTATION AND EXPERIMENTAL RESULTS

The algorithm we describe in Section IV-A is implemented in Java and tested on several examples that we constructed. In this section, we briefly describe the implementation details and present our experimental results.

### A. Implementation

The cross-entropy-based testing tool ConCenter is written in Java. Its structure is reflected in Figure 1, and we briefly describe each part of it below.

- **Instrumenter** is an instrumentation tool that adds callbacks at synchronization points, i.e., immediately before and after each synchronized block.
- **Decider** receives a node  $v$  of the JCG of the program under test and chooses which thread is allowed to run next according to current relative probabilities  $RP(v)$  on the control graph edges.
- **Stopper:** on callbacks from the instrumented code it stops the currently running thread using a mutex designated for this thread. Then, it calls `notify()` on the mutex of the thread that can make the next step (based on Decider’s decision).
- **Evaluator** collects the edges of the JCG traversed by the execution path. At the end of each execution it computes the  $S$  value of the execution.
- **Updater** updates the probability distribution table for the next iteration based on the computations of Evaluator.

During the execution, Decider and Evaluator collect big amounts of data. To minimize the sizes of the memory buffers for this data, it is periodically written to files.

## B. Experimental Results

We performed several experiments on different metrics and different types of bugs. The tests were written in Java version 1.5.0 and executed on a 4 CPU machine 64bit “Dual Core AMD Opteron(tm) Processor 280” with clock rate of 2.4GHz and 1MB cache size each. The total memory of the machine is 8GB. The operation system it runs is GNU/Linux 2.4.21-37.ELsmp.

We constructed several examples of programs with bugs that admit a natural performance function. The programs and bugs introduced in them are as follows.

- 1) Buffer overflow and underflow in a standard producer-consumer program. The test program consists of four threads (2 producers and 2 consumers) running concurrently. The buffer size is 5, and each thread can perform 10 push or pop operations.
- 2) Buffer overflow in a scenario similar to the previous case but with the buffer size 45 and 25 operations for each thread.
- 3) Buffer overflow and underflow similar to test 1 but with modulo parameter.
- 4) Stack overflow in a program with two types of threads,  $A$  and  $B$ , and two threads of each type. The pseudo-code of a thread is presented In Figure 3. Each thread performs 10 operations, and the size of the stack is 36. We note that in this test, buffer overflows are very rare (the probability of experiencing a bufer overflow in a random execution under uniform distribution on edges of the JCG is  $O(1/2^n)$ , where  $n$  is the size of the buffer), since the buffer is filled up only in executions in which thread types continuously alternate.

```

myName = A; // or B
for(i = 0; i < 10; i++)
  synchronized(buffer) {
    if((top of stack)== myName)
      pop();
    else
      push(myName);
  } // end loop

```

Fig. 3. The pseudo-code of the stack overflow example with two types of processes

- 5) Deadlocks. The test program has 10 mutexes and three locking threads. Most of the time, a correct locking policy is enforced (mutexes are numbered and the locking is attempted only in the order of their numbering). However, in a small percentage of the executions, a thread locks mutex 8 before locking mutex 7 and thus can cause deadlock.

In each test, we fine-tuned the parameters  $q$  (the best part of the sample),  $\alpha$  (the smoothing parameter), and the modulo parameter to achieve the fastest convergence to the global maximum. The results of ConCEnter are compared with the results of running ConTest and Java scheduler on the same

<b>Buffer Overflow/Underflow</b>			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	unstable	99%
runtime	15 sec	10 sec for 2000 tests	3 sec
<b>Buffer Overflow with a large buffer</b>			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	from 0.02% to 30%	99%
runtime	20 sec	30 sec for 5000 tests	3 sec
<b>Buffer Overflow with MJCG</b>			
	ConCEnter	ConTest	Java scheduler
% successful executions	100%	unstable	99%
runtime	15 sec	10 sec for 2000 tests	3 sec
<b>Stack Overflow with A and B threads</b>			
	ConCEnter	ConTest	Java scheduler
% successful executions	90%	$\ll 2.5\%$	0%
runtime	75 sec	$< 40$ sec	2 sec
<b>Deadlock</b>			
	ConCEnter	ConTest	Java scheduler
% successful executions	90%	at most 1 out of 5000 tests	$\approx 0\%$
runtime	20 min	300 sec	2 sec

Fig. 4. Experimental results

examples. We summarize the results in Table 4. In all examples ConCEnter converged in less than 20 iterations.

We also studied the influence of various CE method parameters on the performance of ConCEnter on the example of test 3 (stack overflow with  $A$  and  $B$  threads). In our experiments, we did not see a significant influence of  $q$  and  $\alpha$  parameters on the performance of ConCEnter. The conclusion is that the modulo parameter is the parameter that seems to have a crucial role in the performance of ConCEnter. It seems that there is a lower and an upper bound for it, between which ConCEnter performs well. If the modulo parameter is smaller than its lower bound, then MJCG loses too much information and thus the CE method does not converge. If, on the other hand, the modulo parameter is larger than its upper bound, then MJCG is too sparse and CE method converges to a local maximum. We summarize these results in Table 5. The runs that did not converge neither to the global nor to the local maximum, did not converge at all after 40 iterations.

modulo	# samples per iteration	% convergence to	
		global maximum	local maximum
none	100, 200, 400	0	100%
2	200	30%	70%
2	400	15%	0%
3	100	20%	55%
3	200	55%	25%
3	400	75%	10%
4	100	5%	80%
4	200	50%	45%
4	400	85%	5%
8	100, 200, 400, 1000	0	100%

Fig. 5. Influence of modulo parameter

## VI. CONCLUSIONS AND FUTURE WORK

We showed a way to find rare bugs in large programs using the CE method by describing an algorithm that adapts the setting of testing of a program to the setting of the combinatorial optimization problem and performs an iterative procedure of CE method for a given performance function. We listed many interesting (potentially buggy) patterns in programs which have natural performance functions. We implemented a testing tool based on the CE method, called ConCenter. We described the structure of ConCenter and presented experimental results of running it on several patterns. We compared the performance of ConCenter to this of ConTest, an IBM testing tool. In future work, we will apply ConCenter to other patterns. We will also investigate other, more sophisticated ways to use CE method in testing. Mainly, we are interested in the following directions:

- **Fine-tuning ConCenter parameters.** Our experiments show that in many cases choosing the best parameters for a given performance function is hard. It is interesting to devise an automatic method to find optimal parameters.
- **Finding the second best.** Our method converges towards some parts of interleaving space where the performance function is maximized locally or globally. There may be other areas where the performance function is maximized. We would like to develop a method to find these other areas which were not covered in previous executions of the algorithm.
- **Increasing the modulo in a part of the graph.** The parameter that defines the modulo reduction seems to be critical to the convergence of the method and it is hard to find the value which will be optimal to all nodes of the JCG. The best strategy here is probably to define a different modulo value for each node. We guess that this can be done by analyzing the graph of previous executions.
- **Coverage.** As mentioned above, increasing test coverage is one of most widely used methods of program verifica-

tion. In testing, a huge effort is made to create tests which cover more of the program. We intend to use CE method with “find the second best” technique to iteratively create execution paths which increase coverage.

- **Partial Replay.** It is often important to replay a previous execution of the program (for example, in order to reproduce a bug). Unfortunately, full replay is very hard or even impossible to achieve in practice. It seems that using the CE method with the performance function that reaches its maximal value on the execution we are trying to replay can produce an execution that resembles it to a high extend.

## ACKNOWLEDGMENT

We thank Reuven Y. Rubinstein and Andrey Dolgin for helpful discussions on the cross-entropy method.

## REFERENCES

- [1] G. Alon, D.P. Kroese, T. Raviv, and R.Y. Rubinshtein. Application of the cross-entropy method to buffer allocation problem in simulation-based environment. *Annals of Operations Research*, 2004.
- [2] L. Bening and H. Foster. *Principles of verifiable RTL design – a functional coding style supporting verification processes*. Kluwer Academic Publishers, 2000.
- [3] M. Biberstein, E. Farchi, and S. Ur. Fidgeting to the point of no return. In *PDPS*, volume 26-30, April 2004.
- [4] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC and Verilog using predicate abstraction. In *MEMOCODE*, pages 7–16. IEEE, 2004.
- [5] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Verification of SpecC using predicate abstraction. *FMSD*, 30(1):5–28, February 2007.
- [6] T. Homem de Mello and R.Y. Rubinstein. Rare event simulation and combinatorial optimization using cross entropy: estimation of rare event probabilities using cross-entropy. In *Winter Simulation Conference*, pages 310–319, 2002.
- [7] D.L. Dill. What’s between simulation and formal verification? In *35st DAC*, pages 328–329. IEEE Computer Society, 1998.
- [8] U. Dubin. The cross-entropy method for combinatorial optimization with applications. Master Thesis, The Technion, 2002.
- [9] O. Edelstein, E. Farchi, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.
- [10] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *SEA*, Acta Press, 2002.
- [11] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [12] J.M. Keith and D.P. Kroese. Rare event simulation and combinatorial optimization using cross entropy: sequence alignment by rare event simulation. In *Winter Simulation Conference*, pages 320–327. ACM, 2002.
- [13] S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.
- [14] L. Margolin. Cross-entropy method for combinatorial optimization. Master Thesis, The Technion, 2002.
- [15] D. Peled. *Software reliability methods*. Springer-Verlag, 2001.
- [16] G. Ratzaby, B. Sterin, and S. Ur. Improvements in coverability analysis. In *FME*, pages 41–56, 2002.
- [17] R.Y. Rubinshtein. Optimization of computer simulation models with rare events. *European Journal on Operations Research*, 99:89–112, 1997.
- [18] R.Y. Rubinshtein. The cross-entropy method and rare-events for maximal cut and bipartition problems. *ACM Transactions on Modelling and Computer Simulation*, 12(1):27–53, 2002.
- [19] R.Y. Rubinstein and D.P. Kroese. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Information Science and Statistics. Springer-Verlag, 2004.
- [20] E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. Sequential circuit test generation in a genetic algorithm framework. In *DAC*, pages 698–704. ACM/IEEE, 1994.

- [21] D.G. Saab, Y.G. Saab, and J. Abraham. A test cultivation program for sequential VLSI circuits. In *CAV, LNCS*, pages 216–219, 1992.
- [22] S.D. Stoller. Testing concurrent java programs using randomized scheduling. In *RV*, volume 70(4) of *ENTCS*. Elsevier, 2002.
- [23] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design and Test of Computers*, 18(4):36–45, 2001.
- [24] Verisity. Surecove’s code coverage technology. <http://www.verisity.com/products/surecov.html>, 2003.
- [25] J.A. Whittaker. *How to Break Software*. Addison-Wesley, 2003.
- [26] H. Zhu, P.V. Hall, and J.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.

## APPENDIX

Here we present the more formal explanation of CE method. The main ideas behind the CE algorithm are as follows. Let  $\mathcal{X}$  be a space of vectors. Let  $\{f(\cdot; v)\}$  be a family of probability density functions (pdfs) on  $\mathcal{X}$ , where  $v$  is a parameter vector. Let  $S: \mathcal{X} \rightarrow \mathbb{R}^+$  be a function that gives each vector in  $\mathcal{X}$  a non-negative value. In the general version of the CE method, we estimate the probability

$$l = P_u(S(X) \geq \gamma) = E_u I_{\{S(X) \geq \gamma\}} \quad (3)$$

for some  $\gamma \in \mathbb{R}$  - under input  $f(\cdot; u)$  pdf. If this probability is very small, say, smaller than  $10^{-5}$ , we say that  $\{S(X) \geq \gamma\}$  is a rare event. A straightforward way to estimate  $l$  is to draw a random sample  $X_1, \dots, X_N$  from  $\mathcal{X}$  according to  $f(\cdot; u)$ , and then estimate  $l$  by examining the sample. The problem is, that when  $l$  is a rare event, the sample might have to be very large in order to estimate  $l$  accurately. A better way is to draw the sample according to some other probability density function  $g$  that raises the probability of  $S(X) \geq \gamma$ . Using the density  $g$  we can rewrite equation 3 as

$$l = \int I_{\{S(X) \geq \gamma\}} \frac{f(x; u)}{g(x)} g(x) dx = E_g I_{\{S(X) \geq \gamma\}} \frac{f(x; u)}{g(x)}.$$

The  $g$  called *importance sampling* density. An unbiased estimator of  $l$  is

$$\hat{l} = \frac{1}{N} \sum_{i=1}^N I_{\{S(X_i) \geq \gamma\}} \frac{f(X_i; u)}{g(X_i)}, \quad (4)$$

where  $\hat{l}$  called *importance sampling* (IS) estimator.

The best  $g$  to estimate  $l$  is

$$g^*(x) = \frac{I_{\{S(x) \geq \gamma\}} f(x; u)}{l} \quad (5)$$

by using this change of measure in equation 4 we get

$$l = I_{\{S(x) \geq \gamma\}} \frac{f(X_i; u)}{g(X_i)}$$

for all  $i$ . The  $g^*$  function depends on  $l$  which is unknown. It is convenient to choose  $g$  from the  $\{f(\cdot; v)\}$  family. The idea is to choose *tilting parameter*  $v$  such that distance between  $g^*$  and  $f(\cdot; v)$  is minimal. For this purpose the CE method uses Kullback-Leibler “distance” (also called cross-entropy). The Kullback-Leibler “distance” between  $g$  and  $h$  defined as:

$$\mathcal{D}(g, h) = E_g \ln \frac{g(X)}{h(X)} = \int g(x) \ln g(x) dx - \int g(x) \ln h(x) dx.$$

Minimizing the Kullback-Leibler “distance” between  $g^*$  in equation 5 and  $f(\cdot; v)$  is equivalent to solving the maximization problem

$$\max_v \int g^*(x) \ln f(x; v) dx,$$

since the value of  $\int g^*(x) \ln g^*(x) dx$  is constant. Substituting  $g^*$  from equation 5 we obtain equivalent maximization problem

$$\max_v \int \frac{I_{\{S(x) \geq \gamma\}} f(x; u)}{l} \ln f(x; v) dx,$$

which is equivalent to

$$\max_v D(v) = \max_v E_u I_{\{S(x) \geq \gamma\}} \ln f(x; v),$$

and using importance sampling again with a change measure  $f(\cdot; w)$  we can write

$$\max_v D(v) = \max_v E_w I_{\{S(x) \geq \gamma\}} \frac{f(x; u)}{f(x; w)} \ln f(x; v).$$

The optimal solution is

$$v^* = \operatorname{argmax}_v E_w I_{\{S(x) \geq \gamma\}} \frac{f(x; u)}{f(x; w)} \ln f(x; v)$$

and can be estimated by following stochastic program

$$\max_v \hat{D}(v) = \max_v \frac{1}{N} \sum_{i=1}^N I_{\{S(X_i) \geq \gamma\}} \frac{f(X_i; u)}{f(X_i; w)} \ln f(X_i; v), \quad (6)$$

where  $X_1, \dots, X_N$  is a random sample from  $f(\cdot; w)$ .

The CE method in each iteration solves Equation 6 based on the solution of previous iteration. We demonstrate the solution in one iteration by example, which is very similar to the use of the cross-entropy method in the paper.

Let  $\vec{X}$  be a random vector  $(X_1, \dots, X_N) \sim \operatorname{Ber}(p)$ , and parameter  $v = p$ . The probability density function is

$$f(\vec{X}; p) = \prod_{i=1}^N p_i^{X_i} (1 - p_i)^{1 - X_i},$$

and since each  $X_j$  can only be 0 or 1, we have

$$\frac{\partial}{\partial p_j} \ln f(X; p) = \frac{X_j}{p_j} - \frac{1 - X_j}{1 - p_j} = \frac{X_j - p_j}{(1 - p_j)p_j}.$$

We can compute the maximum in Equation 6 by setting the first derivatives with respect to  $p_j$  equal to zero, for  $j = 1, \dots, N$ :

$$\begin{aligned} \frac{\partial}{\partial p_j} \sum_{i=1}^N I_{\{S(X_i) \geq \gamma\}} \ln f(X_i; p) &= \\ &= \frac{1}{(1 - p_j)p_j} \sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}} (X_{ij} - p_j) = 0, \end{aligned}$$

where  $X_{ij}$  is  $X_j$  in sample  $i$ .

Thus, we have

$$p_j = \frac{\sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}} X_{ij}}{\sum_{i=N}^N I_{\{S(X_i) \geq \gamma\}}}.$$

# Automatic Abstraction Refinement for Generalized Symbolic Trajectory Evaluation

Yan Chen\*, Yujing He\*, Fei Xie\* and Jin Yang<sup>†</sup>

\*Department of Computer Science, Portland State University, Portland, OR 97207. {chenyan, hey, xie}@cs.pdx.edu

<sup>†</sup>Validation Research Lab, Intel Corporation, Hillsboro, OR 97124. jin.yang@intel.com

**Abstract**—In this paper, we present AutoGSTE, a comprehensive approach to automatic abstraction refinement for generalized symbolic trajectory evaluation (GSTE). This approach addresses imprecision of GSTE’s quaternary abstraction caused by under-constrained input circuit nodes, quaternary state set unions, and existentially quantified-out symbolic variables. It follows the counterexample-guided abstraction refinement framework and features an algorithm that analyzes counterexamples (symbolic error traces) generated by GSTE to identify causes of imprecision and two complementary algorithms that automate model refinement and specification refinement according to the causes identified. AutoGSTE completely eliminates false negatives due to imprecision of quaternary abstraction. Application of AutoGSTE to benchmark circuits from small to large size has demonstrated that it can quickly converge to an abstraction upon which GSTE can either verify or falsify an assertion graph efficiently.

## I. INTRODUCTION

Symbolic trajectory evaluation (STE) [1]–[3] is a powerful model checking technique based on quaternary symbolic simulation. Within its application domain, STE is often much easier to use and less sensitive to state space explosions, compared to classic symbolic model checking (SMC) techniques [4]–[6]. Despite its efficiency, capacity, and ease-to-use, STE is limited in property expressiveness: properties over infinitely long time intervals cannot be specified and verified using STE.

Generalized symbolic trajectory evaluation (GSTE) [7], [8] represents a significant extension to STE. GSTE supports verification of properties over infinitely long time intervals, namely  $\omega$ -regular properties, thus making it as expressive as classic SMC for linear time logics. Meanwhile, GSTE maintains the efficiency, capacity, and ease-to-use of STE, in particular, GSTE inherits STE’s automatic abstraction techniques.

The key to the high capacity of STE and GSTE is the abstraction based on a quaternary state representation (a.k.a., quaternary abstraction) which, however, is also their weakness. The imprecision of quaternary abstraction may lead to false negatives: STE and GSTE may report the result of X (unknown) instead of 0 or 1. This is worsened by the fixed-point computation of GSTE which introduces additional possibilities for unknowns to creep into verification. As a result, there are three possible causes of abstraction imprecision in GSTE:

1. Under-constrained input circuit nodes.
2. Quaternary state set unions.
3. Existentially quantified-out symbolic variables.

Currently in GSTE, imprecision due to Cause 1 is addressed by manually introducing symbolic constants or variables to

constrain input nodes. Imprecision due to Causes 2 and 3 is eliminated manually by (1) model refinement: introducing boolean variables (a.k.a., precise nodes) to represent the state space more accurately or (2) specification refinement: applying semantics-preserving transformations to the assertion graph [9]. These manual refinements are often quite involved and require in-depth knowledge of the circuit being verified. Furthermore, the level of abstraction determines the verification complexity of GSTE: the more detailed the abstraction, the higher the verification complexity. Therefore, automatic abstraction refinement algorithms that can quickly converge to an appropriate level of abstraction are highly desired.

In this paper, we present AutoGSTE, a comprehensive approach to automatic abstraction refinement for GSTE, which addresses abstraction imprecision due to under-constrained input nodes, quaternary state set unions, and quantified-out symbolic variables. It follows the counterexample-guided abstraction refinement framework and features an algorithm that analyzes the counterexample (symbolic error trace) generated by GSTE to identify causes of imprecision and two complementary algorithms that automate model refinement and specification refinement according to the causes identified.

- The analysis algorithm identifies these causes by backtracking through the counterexample and conducting fan-in analysis over the circuit.
- If imprecision is due to under-constrained input nodes, symbolic constants are introduced on non-loop edges of the assertion graph while symbolic variables are introduced on loop edges of the assertion graph.
- Using model refinement, the circuit nodes which obtain the unknown values due to imprecision caused by quaternary state set unions or quantified-out symbolic variables are identified and marked as precise nodes in the circuit.
- Using specification refinement, according to the causes identified, the appropriate kind of semantic-preserving transformation is applied to the assertion graph: for imprecision due to quaternary state set unions, loop-unrolling is applied and for imprecision due to quantified-out symbolic variables, case-splitting is applied.

We have implemented AutoGSTE in the Intel *Forte* environment [10] and upon GSTE, and applied it to benchmark circuits from small to large size. The experiments demonstrate that AutoGSTE can quickly converge to an abstraction upon which GSTE can verify or falsify an assertion graph efficiently.

**Related Work.** There has been much research on abstraction refinement for model checking of both hardware and software. Space limitation precludes a detailed discussion. Counterexample-guided abstraction refinement is a well-known methodology in model checking to combat the state space explosion problem [4], [11], [12]. Particularly relevant to our research is the work on abstraction refinement for STE. In [13], symbolic constants are automatically introduced to constrain under-constrained input nodes of circuits based on counterexamples from STE. In [14], a SAT-based algorithm was developed to assist manual refinement of STE assertions. To our best knowledge, our approach is the first automatic abstraction refinement framework for GSTE that completely eliminates false negatives caused by abstraction imprecision.

The rest of this paper is organized as follows. In Section II, we introduce the basics of STE and GSTE and the quaternary abstraction that they employ. In Section III, we discuss in detail the potential causes for imprecision of the quaternary abstraction. In Section IV, we present AutoGSTE, our approach to automatic abstraction refinement. In Section V, we evaluate our algorithms on small to large size benchmark circuits. In Section VI, we conclude this paper and touch on future work.

## II. BACKGROUND

In this section, we introduce a simple circuit model upon which STE and GSTE were developed, the basics of STE and GSTE, and their quaternary abstraction. For more details on STE and GSTE, we refer the readers to [1], [2], [7], [8].

### A. A Simple Circuit Model

A circuit  $M$  consists of a set of boolean nodes  $N$ . A state is an assignment to all the nodes in  $N$ . The node set  $N$  can be partitioned into two disjoint sets: *state nodes*  $N_S$  and *input nodes*  $N_I$ . There is a *next-state function*  $\chi_n(N)$  for each state node  $n \in N_S$ . The set of next state functions defines how the circuit transitions between states. The transition can also be defined by the equivalent *transition relation*  $R(N, N') = \bigwedge_{n \in N_S} (n' = \chi_n(N))$ , where  $N'$  is a copy of  $N$  to hold the values for  $N$  after the transition and  $n' \in N'$  is the copy of  $n$ . A *trace* of the circuit is a state sequence  $\sigma = [s_0, s_1, s_2, \dots]$  such that for every index  $i$ ,  $s_{i+1}(n) = \chi_n(s_i(N))$  for every state node  $n \in N_S$  and  $s_{i+1}(n)$  is unconstrained for every input node in  $n \in N_I$ .

### B. Symbolic Trajectory Evaluation

An STE assertion can be viewed as a labeled linear graph representing a finite time line. Each edge in the graph represents a time unit and is labeled with two sets of circuit states (or equivalently state predicates), one of which is called the *antecedent label* and the other the *consequent label*.

The symbolic simulation algorithm in STE starts from the first edge  $(v_0, v_1)$  in the graph and computes the set of states  $sim(v_0, v_1)$  simulated by the edge. In this case, it is the set of states satisfying the antecedent label on the edge. The algorithm then performs a single simulation step by computing the *post-image*,  $post(sim(v_0, v_1))$ , of this simulation state set

and intersects the result with the set of states satisfying the antecedent label on the second edge. The result is the set of states  $sim(v_1, v_2)$  simulated by the second edge. The post-image of a state set  $S(N)$ , denoted by  $post(S(N))$ , is given by  $\exists N^-. S(N^-) \wedge R(N, N^-)$ , i.e., the set of states reachable from a state in  $S(N)$  in a single state transition. The algorithm repeats this step until it reaches the last edge in the graph. Once the simulation is complete, the consequent label on each edge is checked against the set of states simulated by this edge to see if it is satisfied, i.e., the set of states simulated by the edge is a subset of states allowed by the consequent predicate.

### C. Generalized Symbolic Trajectory Evaluation

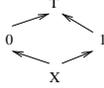
A *GSTE assertion graph* is defined as a quintuple  $G = (V, v_0, E, ant, cons)$ , where  $V$  is a set of *vertices*,  $v_0$  is the *initial vertex*,  $E$  is a set of *directed edges*, and *ant* and *cons* are functions that map each edge to an antecedent label and a consequent label, respectively. Every finite path in the GSTE assertion graph from the initial vertex is an STE assertion graph. Therefore, a circuit  $M$  satisfies a GSTE assertion graph  $G$  if it satisfies every STE assertion graph derived from the GSTE assertion graph by following a finite path from the initial vertex in the graph. More formally, the circuit satisfies the assertion graph, denoted by  $M \models G$ , if for every boolean assignment to all the symbolic constants, every finite path in the graph, and every finite state trace in the circuit of the same length, the trace satisfies the antecedent sequence on the path implies it also satisfies the consequent sequence on the path.

To model check a GSTE assertion graph against a circuit, a symbolic simulation is performed to compute a set of states simulated by each edge in the graph. A state is in this set if there is a finite path from the initial vertex leading to the edge and a finite trace leading to the state such that the trace satisfies the sequence of the antecedent labels along the path. Obviously, for an edge coming out of the initial vertex, any state satisfying the antecedent label on the edge is simulated by the edge. Further, for an edge  $e$  and a successor edge  $e'$  of  $e$ , if  $s$  is a state simulated by  $e$ , then any next state  $s'$  of  $s$  that satisfies the antecedent label on  $e'$  is simulated by  $e'$ . This leads to a GSTE model checking algorithm with an iterative symbolic simulation phase. Since an assertion graph may contain loops and an edge may be reached from the initial vertex through different paths, the algorithm requires a form of least fixed-point computation in the simulation phase [7].

### D. Quaternary Abstraction

When the circuit becomes large, the likelihood for a symbolic model checking algorithm to encounter the *state explosion* problem increases drastically. To overcome the problem, some sort of abstraction must be applied to the circuit. In STE, it is the quaternary modeling of the circuit where each node has the four quaternary values  $\{0, 1, X, \top\}$  instead of the two boolean values. There is a partial information order among the four quaternary values as shown in Figure 1 (a). For instance,  $X$  contains less information than either 0 or 1. Both 0 and 1 contain less information than  $\top$ , which represents the

over-constrained value. Besides the quaternary generalization of the boolean operations, two new operations are defined: the greatest lower bound  $\sqcap$  and the least upper bound  $\sqcup$  of any two quaternary values. Figure 1 (b) lists the truth tables for the basic quaternary operations.



(a) quaternary values and information ordering

$\&$	X	0	1	T		X	0	1	T		$\sqcup$	X	0	1	T		$\sqcap$	X	0	1	T	
X	X	0	X	T	X	X	X	1	T	X	X	X	0	1	T	X	X	X	X	X	X	X
0	0	0	0	T	0	X	0	1	T	0	1	0	0	0	T	0	X	0	X	0	0	X
1	X	0	1	T	1	1	1	1	T	1	0	1	1	T	1	1	X	1	1	1	X	X
T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	X	0	1	1	T	X

(b) quaternary operations

Fig. 1. Quaternary Operations

Given such a quaternary abstraction, any state set in the circuit can be represented either precisely or approximately by a quaternary assignment to the nodes in the circuit. A node has a boolean value in the quaternary assignment if it has the same boolean value in every state of the state set. Otherwise, it has value  $X$ . The empty set is represented by assigning  $\top$  to one or more variables depending on where the conflict occurs in the circuit. For instance, consider a circuit with three nodes  $p$ ,  $q$  and  $r$ . The quaternary assignment for the singleton state set  $\{[p=1, q=0, r=1]\}$  is  $[p=1, q=0, r=1]$ , and the assignment for the state set  $\{[p=1, q=0, r=1], [p=1, q=1, r=1]\}$  is  $[p=1, q=X, r=1]$ .

With this abstraction, the state space becomes much smaller. In general, for a circuit with  $n$  boolean nodes, there are  $2^n$  different states and thus  $2^{2^n}$  different sets of states. On the other hand, there are only  $4^n$  different quaternary assignments. Furthermore, all the operations become much more efficient in the quaternary abstraction. For instance, the intersection  $\cap$  of two state sets becomes a bit-wise  $\sqcap$  of the two corresponding quaternary assignments, and the union  $\cup$  becomes the bit-wise  $\sqcup$ . The post-image function becomes the bit-wise quaternary generalization of the next state functions for the state nodes together with  $X$  for the input nodes. Instead of mapping a state set to another state set, it maps the quaternary assignment representing the first state set to the one representing the second state set.

A problem with the scalar quaternary model, though, is that it is often too coarse. This could easily lead to false negative verification results, since many types of constraints among nodes are lost in the abstraction. To overcome this problem, STE uses a technique called *symbolic indexing* to encode a set of quaternary assignments. For instance, for the circuit with three nodes  $p$ ,  $q$  and  $r$ , consider the following set of boolean assignments

$$\{[p=0, q=1, r=1], [p=1, q=0, r=0], [p=1, q=0, r=1]\}.$$

It captures two basic boolean constraints: (1)  $p$  and  $q$  are

mutual exclusive, and (2)  $q$  is true implies  $r$  is true. These two constraints are completely lost in the single quaternary assignment corresponding to the set  $[p=X, q=X, r=X]$ . However, using a symbolic constant  $C$ , the two constraints can be precisely encoded in the following symbolic indexing expression

$$(!C \rightarrow [p=0, q=1, r=1]) \wedge (C \rightarrow [p=1, q=0, r=X]),$$

which means that when  $C$  is 0 then the first quaternary is chosen, and when  $C$  is 1 then the second is chosen. For simplicity, we omit  $X$ 's in the quaternary assignments below.

When verifying an STE assertion against a circuit, any boolean constraint in the STE assertion can be expressed as a symbolic quaternary assignment using symbolic constants in order to drive the symbolic quaternary simulation in STE. Furthermore, the abstract symbolic quaternary simulation of the circuit can be made into the precise symbolic boolean simulation by assigning enough symbolic constants to the circuit nodes in the antecedent of the assertion. However, the same is no longer true in GSTE for assertion graphs with loops. To address this problem, GSTE allows introducing symbolic variables into symbolic quaternary assignments. Unlike a symbolic constant, a *symbolic variable* is a boolean variable that can change its value, and is existentially quantified out after every single step simulation. One way to look at the difference between symbolic variables and symbolic constants is that symbolic variables symbolically index a set of edges with scalar values while symbolic constants symbolically index a set of assertion graphs with scalar values.

### III. IMPRECISION OF QUATERNARY ABSTRACTION

#### A. Three Causes of Quaternary Abstraction Imprecision

There are three causes of imprecision in GSTE's quaternary abstraction: one inherited from STE and the other two new in GSTE. In this section, we elaborate on these causes and present a simple illustrative example.

1) *Under-constrained input circuit nodes*: In (G)STE, the input nodes of a circuit are constrained by the antecedent on each edge of an assertion graph. If an input node is unconstrained, it will be assigned the value of  $X$ . Such  $X$ 's may cause consequent violations in the symbolic simulation.

2) *Quaternary state set union*: In the fixed-point computation of GSTE, a set of circuit states simulated by each edge in the assertion graph is computed. The set of states is represented approximately by a quaternary assignment and the union of two sets of states is approximated by the bit-wise  $\sqcup$  of the two corresponding quaternary assignments. Such unions may introduce additional  $X$ 's into the verification.

3) *Existentially quantified-out symbolic variables*: In GSTE, a symbolic variable is a boolean variable that can change its value every time the corresponding edge is visited, and is existentially quantified out after every single step simulation. So after leaving the edge with symbolic variables, circuit nodes associated with these variables may become  $X$ .

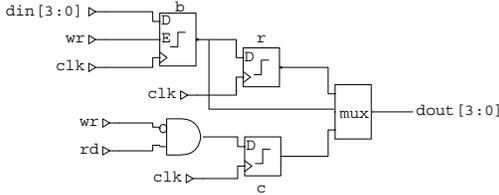


Fig. 2. Buffered Register

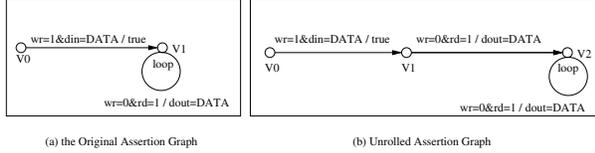


Fig. 3. Assertion Graph for Buffered Register

TABLE I  
QUATERNARY SIMULATION

	(V0, V1)	(V1, V1)
0	[din=DATA, wr=1]	TOP
1	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]
2	[din=DATA, wr=1]	[b=DATA, rd=1, wr=0]
3	[din=DATA, wr=1]	[b=DATA, rd=1, wr=0]

### B. A Simple Illustrative Example of Imprecision

Figure 2 shows a simple buffered-register circuit: the input to the register is first written into a buffer and then transferred into the register in the next clock cycle after the write. Figure 3(a) shows an assertion graph capturing a property to be verified on the circuit: after a write, if there are only read requests, the output is always the data accompanying the write. We run GSTE to verify the circuit against the assertion graph. The step-by-step result of the quaternary simulation is listed in Table I. GSTE reports a violation of the consequent  $dout = DATA$  on the loop edge  $(V1, V1)$  due to an  $X$  assigned to  $dout$ . However, it is not difficult to observe that the property holds on the circuit. Therefore, GSTE has reported a false negative. The quaternary assignment of  $(V1, V1)$  in Iteration 2,  $[b = DATA, rd = 1, wr = 0]$ , is obtained from the union of the assignment of  $(V1, V1)$  in Iteration 1,  $[b = DATA, c = 0, rd = 1, wr = 0]$ , and the assignment representing the set of new reachable states in Iteration 2,  $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$ . This union introduces additional  $X$ 's, which lead to the  $X$  value of  $dout$ .

## IV. ABSTRACTION REFINEMENT

In this section, we start by reviewing the state-of-art manual abstraction refinement for GSTE and then present AutoGSTE, our approach to automatic abstraction refinement for GSTE. First, we give its overarching framework. Then, we introduce our algorithm for identifying causes of abstraction imprecision from counterexamples generated by GSTE. After that, we discuss how we automate abstraction refinement according to the causes identified. Finally, we discuss the correctness of AutoGSTE.

TABLE II  
QUATERNARY SIMULATION

	(V0, V1)	(V1, V2)	(V2, V2)
0	[din=DATA, wr=1]	TOP	TOP
1	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	TOP
2	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	[b=DATA, r=DATA, c=1, rd=1, wr=0]
3	[din=DATA, wr=1]	[b=DATA, c=0, rd=1, wr=0]	[b=DATA, r=DATA, c=1, rd=1, wr=0]

### A. Manual Abstraction Refinement

Currently in GSTE, the imprecision due to quaternary state set unions and quantified-out symbolic variables is addressed with two manual strategies. The first strategy is to manually mark a set of circuit nodes as *precise nodes* [9]. For these nodes, by using the parametric representation, their values and the relationships among them are always represented exactly by using boolean expressions as their values. In the above example, we can mark the node  $c$  as a precise node. This will prevent the union of  $[b = DATA, c = 0, rd = 1, wr = 0]$  and  $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$ . We rerun GSTE to verify the circuit with the precise node against the original assertion graph in Figure 3(a). GSTE reports that the assertion graph holds on the circuit.

The second strategy is to manually apply semantics-preserving transformations to the assertion graph. Typical transformations include case-splitting of an edge and unrolling of a loop. In the case of the above example, we can apply loop-unrolling to the assertion graph in Figure 3(a) to obtain a semantically equivalent assertion graph shown in Figure 3(b). We run the circuit against the refined assertion graph and the step-by-step simulation is listed in Table II. In essence, the loop-unrolling prevents the union of  $[b = DATA, c = 0, rd = 1, wr = 0]$  and  $[b = DATA, r = DATA, c = 1, rd = 1, wr = 0]$ , which are now associated with two separate edges  $(V1, V2)$  and  $(V2, V2)$ , respectively. Intuitively, this refines the assertion graph to mimic the real computation flow of the circuit.

Both strategies above require manual efforts. To apply these strategies, one must have in-depth understanding of the circuit being verified and be able to identify where the imprecision is introduced by GSTE. Therefore, it is highly desired that the causes for imprecision can be automatically identified and these manual strategies can be automated. (For more discussion of the manual strategies, we refer readers to [9]).

### B. AutoGSTE: Automatic Abstraction Refinement

AutoGSTE employs a counterexample-guided abstraction refinement loop formed by GSTE, the counterexample analysis algorithm, and the abstraction refinement algorithms, as shown in Figure 4. GSTE applies quaternary abstraction to verify a circuit  $M$  against an assertion graph  $G$ . If GSTE reports an assertion violation: a consequent in the assertion graph is violated, it generates a counterexample (a symbolic error trace), which is a sequence of simulation steps that lead

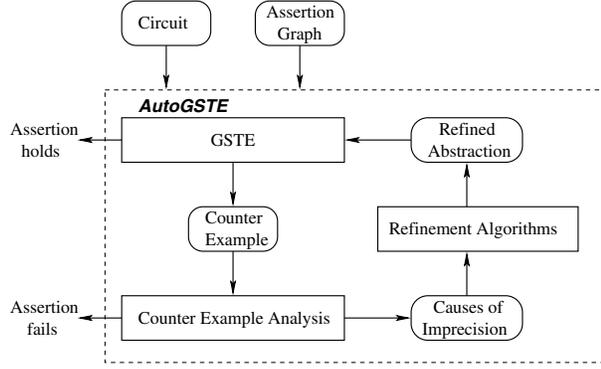


Fig. 4. Automatic Abstraction Refinement Loop for GSTE

to the consequent violation. The counterexample analysis algorithm exams the counterexample to identify the causes of the consequent violation. If it is caused by conflicting values to certain circuit nodes, the assertion fails. Otherwise, the refinement algorithms conduct abstraction refinement according to the causes identified. The model refinement algorithm automatically identifies precise nodes that need to be marked in the circuit. The specification refinement algorithm applies semantics-preserving transformations to certain edges of the assertion graph on-the-fly in the GSTE symbolic simulation.

### C. Identification of Causes for Consequent Violation

1) *Counterexample Generation from GSTE*: When GSTE reports an assertion failure, it also presents a computing history which is a symbolic searching tree of triples  $(e, src, dest)$ , where  $e$  is an edge that GSTE simulated, and  $src$  and  $dest$  are the quaternary states before and after the simulation of the edge  $e$ . Given the triple which causes the consequent violation, we backtrack through the computing history to find the occurrence of the triple  $(e', src', src)$ , and then continue the backtracking until we reach an edge originated from the initial vertex in the assertion graph. The sequence of triples traversed during the backtracking forms a counterexample  $CE$  that leads to the consequent violation. Formally, we define the counterexample as a sequence of triples  $CE = [t_1, t_2, \dots, t_l]$  such that for every index  $1 \leq i \leq l$ ,  $t_i = (edge_i, src_i, dest_i)$ , where  $edge_i$  is the edge traversed in step  $i$  and  $src_i$  and  $dest_i$  are the states before and after this simulation of  $edge_i$ .

2) *Identification of Causes*: Once the counterexample is generated, the counterexample analysis algorithm is applied to determine the causes of the consequent violation. If the violation is due to conflicting values to certain circuit nodes in a circuit state and a corresponding consequent, we know the circuit does not satisfy the assertion graph. If the violation is caused by the unknown value of a circuit node, starting from the circuit node, our algorithm backtracks through the error trace and conducts fan-in analysis over the circuit to identify the circuit nodes where the unknown values were introduced.

The analysis algorithm is shown in Figure 5. It inputs the counterexample  $CE$ , the post-image function  $post$  and

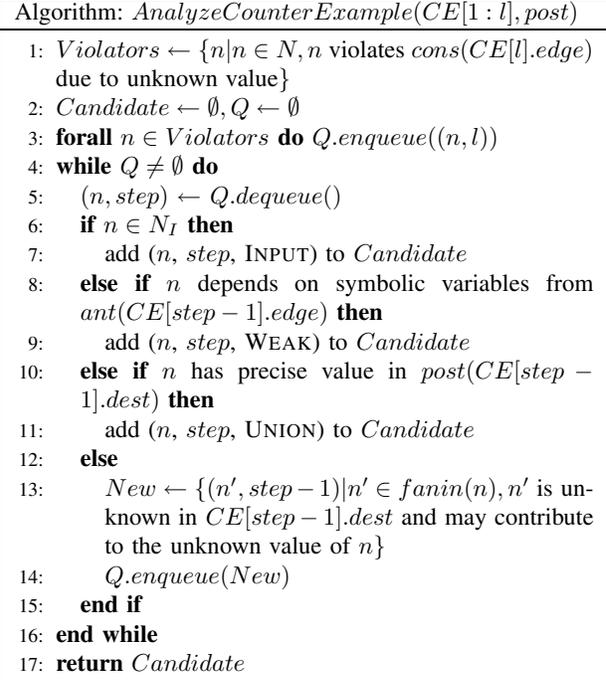


Fig. 5. Counterexample Analysis Algorithm

outputs a set of circuit nodes  $Candidate$  which get unknown values directly due to inputs, quaternary state set unions, and quantified-out symbolic variables. The cause for the unknown value is attached to each node. In this algorithm,  $fanin(n)$  is the function that identifies all circuit nodes that affect the value of a circuit node  $n$ .

Given the counterexample  $CE$ , we start with the time step  $l$  at which a consequent violation is reported. We first decide which circuit nodes have unknown values and cause the violation. This can be done by comparing the quaternary assignment representing the set of states simulated by the edge,  $CE[l].dest$ , and the quaternary assignment representing the consequent of the edge,  $cons(CE[l].edge)$ . If a circuit node  $n$  has an unknown value in  $CE[l].dest$ , while having a boolean value in  $cons(CE[l].edge)$ , we know that  $n$  violates  $cons(CE[l].edge)$ . All violating circuit nodes are put into a queue  $Q$  (Steps 1-3).

For each node  $n$  in  $Q$ , if  $n$  is an input node, we identify  $n$  and mark the cause as “INPUT” (Steps 4-7). If  $n$  depends on a symbolic variable in  $ant(CE[l-1].edge)$ , we identify  $n$  and mark the cause as “WEAK” (Steps 8-9). If  $n$  has precise value in  $post(CE[l-1].dest)$ , we identify  $n$  and mark the cause as “UNION” (Steps 10-11). All the identified circuit nodes are added into the node-cause set  $Candidate$ .

If none of the above conditions holds, we backtrack to the previous time step  $l-1$ . We also conduct a one-level fan-in analysis from  $n$  in the circuit to identify all nodes that affect  $n$ . Among these nodes, for each node  $n'$  with an unknown value in the quaternary assignment of  $CE[l-1].dest$  and that may

contribute to the unknown value of  $n$  in  $CE[l].dest$ , we repeat this analysis for  $n'$  until one of the three causes above is found in the counterexample, by putting  $n'$  into  $Q$  (Steps 13-14). We improve the accuracy of this analysis through examining the different types of gates case-by-case.

It is possible that at the same time, a circuit node can get unknown value due to both a state set union and a quantified-out symbolic variable. In our refinement loop, one cause is identified at a time and eventually both causes are identified.

Using the parametric representation, a circuit node can have a partially unknown value, i.e., under certain condition, node  $n$  has an unknown value. In the backtracking, we keep track of the condition under which node  $n$  has an unknown value. In the fan-in analysis of  $n$ , the condition is propagated to  $n'$  the same way as backward reasoning in bidirectional GSTE [8].

#### D. Refinement Algorithms

1) *Constraining Input Nodes*: If a circuit node identified by the analysis algorithm is an input node, a symbolic constant or variable is introduced in the antecedent of the corresponding edge of the assertion graph to rule out the unknown value of this node. In STE, this type of imprecision can be addressed by constraining the inputs using symbolic constants. In [13], an approach to automatically constraining the input nodes with symbolic constants has been proposed for STE. The same heuristics can be applied in our approach. However, in GSTE, constraining symbolic constants on loop edge may cause the input nodes have the same input signals every time the edge is simulated. Therefore, if the edge is on a loop, a symbolic variable should be introduced.

2) *Marking Precise Nodes*: After we identify the circuit nodes (i.e., the nodes in *Candidate*) that get unknown values due to quaternary state set unions or quantified-out symbolic variables, we can refine the circuit model  $M$  by marking these nodes as precise nodes, and then rerun GSTE with the refined model. The correctness of this refinement is guaranteed by Lemma 4.1. Let  $precise(M, p)$  be the circuit model with the circuit node  $p$  marked as a precise node in  $M$ .

*Lemma 4.1*: (Yang and Seger [9])

Given  $G, \forall p \in N_S(M), M \models G$  iff  $precise(M, p) \models G$ .

There are two strategies in marking the identified nodes: we can mark them either all at once or one at a time. It is easy to see that there is a trade off between these two strategies. The first strategy may converge faster while it may lead to more precise nodes than necessary. The second approach may find a smaller set of precise nodes while it may require more iterations to converge. Heuristics can be applied to improve these two strategies, for instance, an effective heuristic in practice is to pick control nodes as precise nodes over data nodes when using the second strategy.

3) *On-the-fly Transformations of Assertion Graphs*: Making too many nodes precise may cause state space explosions. And identifying a minimal set of circuit nodes as precise nodes is challenging. An alternative approach to address imprecision caused by unions or symbolic variables is to conduct on-the-fly semantics-preserving transformations to the assertion graph

Algorithm: <i>ExtendedGSTE</i> ( $G, post, Edges$ )	
1:	<b>for all</b> $e$ from $v_0$ <b>do</b> $Q.enqueue((e, ant(e)))$
2:	<b>for all</b> $e \in Edges$ <b>do</b> $Hash(e) \leftarrow \emptyset$
3:	<b>while</b> $Q \neq \emptyset$ <b>do</b>
4:	$(e', sim(e')) \leftarrow Q.dequeue()$
5:	<b>for all</b> successor edge $e$ of $e'$ <b>do</b>
6:	$NewState \leftarrow post(sim(e')) \cap ant(e)$
7:	<b>if</b> $e \in Edges$ <b>then</b>
8:	<b>if</b> $e$ is marked UNION <b>then</b>
9:	<b>if</b> $NewState \notin Hash(e)$ <b>then</b>
10:	$Hash(e) \leftarrow Hash(e) \cup \{NewState\}$
11:	$Q.enqueue((e, NewState))$
12:	<b>end if</b>
13:	<b>else if</b> $e$ is marked WEAK <b>then</b>
14:	$Weak \leftarrow \{\text{states derived from } NewState$ by assigning all combination of boolean values for symbolic variables in $ant(e)\}$
15:	<b>for all</b> $s \in Weak$ <b>do</b>
16:	<b>if</b> $s \notin Hash(e)$ <b>then</b>
17:	$Hash(e) \leftarrow Hash(e) \cup \{s\}$
18:	$Q.enqueue((e, s))$
19:	<b>end if</b>
20:	<b>end for</b>
21:	<b>end if</b>
22:	<b>else</b>
23:	$sim(e) \leftarrow sim(e) \cup NewState$
24:	<b>if</b> there is a change in $sim(e)$ <b>then</b> $Q.enqueue(e, sim(e))$
25:	<b>end if</b>
26:	<b>end for</b>
27:	<b>end while</b>
28:	<b>for all</b> $e \in V$ <b>do</b> consequent check

Fig. 6. GSTE Extended with Semantic-Preserving Transformation

$G$ . The motivation is that in some cases, the circuit nodes only need to keep precise values on some assertion graph edges and making them precise in the whole simulation is too costly.

Our algorithm for on-the-fly transformation of assertion graphs, shown in Figure 6, extends the basic GSTE algorithm to support dynamic loop-unrolling and case-splitting of assertion graph edges in the symbolic simulation. Here  $Edges = \bigcup_{(n,i) \in Candidate} \{CE[i].edge\}$ , namely the set of assertion edges that need to be transformed. The algorithm includes two parts: the on-the-fly transformation (Steps 7-21) and the normal GSTE fixed-point computation (Steps 22-25). In the transformation, to keep precise states, we built a hash table for each edge  $e$  in  $Edges$  (Steps 1-2). When a new post-image  $NewState$  of edge  $e$  is generated, we first check if edge  $e$  is marked as UNION. If yes, we exam the hash table. If  $NewState$  is not in the hash table,  $NewState$  is added to the hash table and the simulation continues with  $NewState$ ; otherwise, a fixed point is reached (Steps 8-12). In essence, this realizes loop-unrolling: Loops in  $G$  are expanded to mimic the real computation flow of the circuit. The correctness of this

loop-unrolling transformation is guaranteed by Lemma 4.2. Let  $l$  be a set of edges forming a loop in  $G$  and  $unroll(G, l)$  be the assertion graph with  $l$  unrolled in  $G$ .

*Lemma 4.2:*

Given  $M$ , for any loop  $l$  in  $G$ ,  
 $M \models G$  iff  $M \models unroll(G, l)$ .

The key intuition of the proof for Lemma 4.2 is as follows. For any finite path in  $G$ , there is one and only one finite path in  $unroll(G, l)$  with the same length and labels, and vice versa. (Space limitation precludes presentation of detailed proofs for Lemma 4.2 and the lemmas and theorems below.)

If edge  $e$  is marked as WEAK, we generate all possible combinations of boolean assignments to the symbolic variables in  $ant(e)$ , and apply these combinations to  $NewState$  to get a set of states  $Weak$ . Each state in  $Weak$  which is not reached before is put into the queue (Steps 13-21). This is equivalent to case splitting of certain edges. The correctness of this transformation is guaranteed by Lemma 4.3. Let  $split(G, e)$  be the assertion graph with edge  $e$  case-split in  $G$ .

*Lemma 4.3:*

Given  $M$ ,  $\forall e \in E(M), M \models G$  iff  $M \models split(G, e)$ .

The key intuition of the proof for Lemma 4.3 is as follows. As  $e$  is case-split into  $e_1, \dots, e_k$ , where  $ant(e_1) \cup \dots \cup ant(e_k) = ant(e)$  and  $cons(e_1) = \dots = cons(e_k) = cons(e)$ . In  $split(G, e)$ ,  $e_1, \dots, e_k$  cover all the possible cases which are covered by  $e$  in  $G$ , without introducing any new cases.

### E. Correctness of AutoGSTE

The abstraction-refinement loop terminates when either GSTE reports verification success or it reports a consequent violation due to conflicting values to certain circuit nodes. The termination of this loop is guaranteed by Theorem 4.1.

*Theorem 4.1:*

$AutoGSTE(M, G)$  terminates.

The basic idea for proving this theorem is that the circuit to verify is finite-state. Our refinement algorithms add input node constraints, mark precise nodes, and apply semantics preserving transformations in a monotonic fashion. The correctness of whole  $AutoGSTE$  immediately follows from Lemma 4.1, Lemma 4.2, Lemma 4.3, and Theorem 4.1.

*Theorem 4.2:*

$M \models G$  iff  $AutoGSTE(M, G)$  returns *true*.

## V. EXPERIMENTAL RESULTS

We have implemented AutoGSTE in the Intel *Forte* environment [10] and upon GSTE. We have conducted experiments on a family of benchmark FIFO circuits from Intel, and analyzed how our approach scale with the depth of FIFO. Figure 7 shows a simple stationary 3-entry 8-bit FIFO circuit. The assertion graph for this circuit is shown in Figure 8. The assertion graph checks whether the empty and full signals of the FIFO circuit are set correctly. The assertion graph is independent of the circuit implementation and data width, and exposes imprecision of quaternary abstraction due to both state set

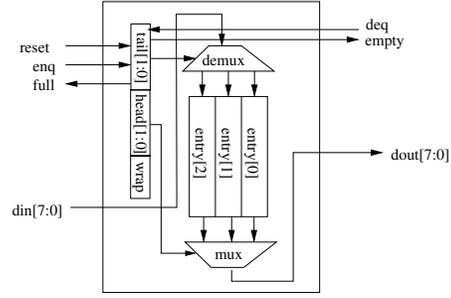


Fig. 7. Stationary FIFO Implementation

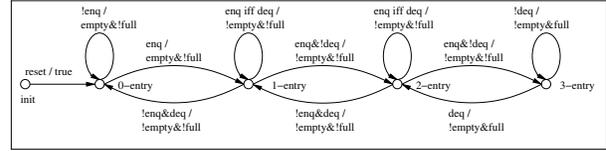


Fig. 8. FIFO Assertion Graph

TABLE III  
MODEL REFINEMENT RESULTS FOR 8-BIT FIFOs

FIFO Depth	Circuit # of Nodes	Mark precise nodes all at once					Mark precise nodes one at a time				
		# of Iter.	# of P. Nodes	Time (Sec.)	BDD Nodes	Mem (MB)	# of P. Nodes.	Time (Sec.)	BDD Nodes	Mem (MB)	
3	181	1	5	0.12	10232	17	3	0.26	8996	17	
8	296	1	7	0.4	32923	19	4	0.81	26708	18	
16	476	1	9	1.1	72189	22	5	2.37	58250	22	
24	787	1	11	2.38	131236	24	6	6.83	104246	24	

unions and quantified-out symbolic variables. All experiments were conducted on a workstation with 3GHz Intel® Xeon® processor with 2GB memory, and all verifications were done on the original circuits with no prior abstraction.

Table III lists the verification results for model refinement. In [9], manual analysis showed the imprecision for the stationary FIFO implementation is caused by different values in the head and tail pointers as well as the wrap bit. Our counter example analysis algorithm identified the same set of potential circuit nodes. If we mark these circuit nodes all at once, only one refinement iteration is needed, and it makes all the non-data-path elements in the circuit precise, which has the same effect as manual effort. If we mark precise nodes one at a time, more iterations are needed. However, interestingly, it finds a smaller set of precise nodes than manual analysis. That is we only need to make the head pointer and the first bit of tail pointer precise. This leads to a smaller number of BDDs generated, which takes less memory than the first strategy. Therefore, it is reasonable to conclude the first strategy requires fewer iterations but more BDDs, while the second strategy is more likely to converge into a smaller set of precise nodes but take more iterations. In practice, there is no definite evidence that which strategy would terminate more quickly since in the first strategy, each iteration takes more time and in the second strategy, more iterations are needed.

Table IV lists the verification results for specification refinement. In the stationary implementation, every assertion

TABLE IV  
SPECIFICATION REFINEMENT RESULTS FOR 8-BIT FIFOs

Circuit FIFO Depth	GSTE on Original assertion graph					Semantic-Preserving Transformation				
	# of Edges	Time (Sec.)	BDD Nodes	Mem (MB)	Result	# of Edges	Time (Sec.)	BDD Nodes	Mem (MB)	Result
3	11	0.01	5	17	Unknown	31	0.23	6	17	Pass
8	26	0.02	5	17	Unknown	201	2.69	6	19	Pass
16	50	0.04	5	17	Unknown	785	17.31	6	26	Pass
24	74	0.07	5	17	Unknown	1753	54.23	6	39	Pass

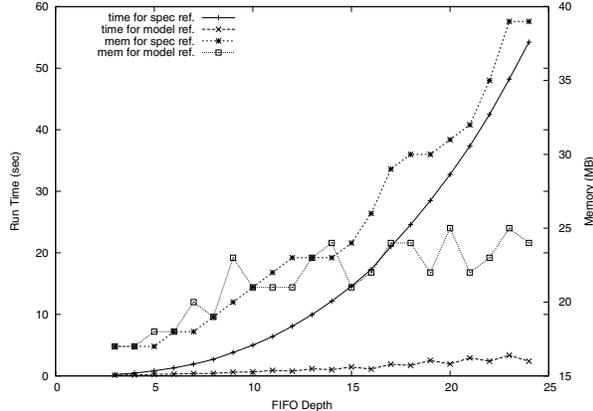


Fig. 9. Time and Memory for Verification of FIFOs

graph edge corresponds to several different combinations of the head and tail pointer values, so the refined assertion graphs grow quickly as the depth of FIFO increases. It is very difficult to conduct this refinement manually [9]. Our counterexample analysis algorithm identified all the edges to be refined, and our refinement algorithm applies the appropriate transformations. The transformations eliminate all the UNION and WEAK cases without increasing the number of BDDs needed. Different from the model refinement, here the memory usage is largely dependent on the number of actual assertion graph edges generated rather than the number of BDDs.

We plot the time and space complexity data of both model refinement and specification refinement in Figure 9. The time and space complexities for model refinement grow almost linearly while those of the specification refinement grow exponentially, with respect to the depth of FIFO. The reason for the later case is that the symbolic variable on every loop edge will split every state into two states carrying different values for the symbolic variable, the verification complexity is determined by the stationary implementation itself. In this experiment, the model refinement approach is more favorable than the specification refinement approach. However, in some cases, the circuit nodes only need to keep precise values on certain edges, or the number of different states simulated by these edges is small, specification refinement would be more efficient than model refinement, and is less likely to suffer from state space explosions. Furthermore, the specification refinement reveals the real computation flow of the circuit, which allows the initial specification to be very high level, and provides a good guidance for debugging in practice.

As we can observe, the verification time for our automatic refinement approach is fairly small, and the amount of memory

used is of reasonable size. None of the two kinds of refinement is easy to conduct manually as the sizes of the circuit and assertion graph increase. Our experiments demonstrate the correctness and effectiveness of our approach.

## VI. CONCLUSIONS

In this paper, we have presented AutoGSTE, a comprehensive approach to automatic abstraction refinement for GSTE. It completely addresses the imprecision of GSTE's quaternary abstraction caused by under-constrained input circuit nodes, quaternary state set join, and quantified-out symbolic variables. Its application to small to large size circuits has demonstrated that it is able to quickly converge to an abstraction upon which GSTE can either verify or falsify an assertion graph efficiently.

Regarding model refinement, further research is needed to explore effective methods for determining a minimal set of circuit nodes as precise nodes in order to minimize the state space that has to be explored. Regarding specification refinement, further research is needed to develop heuristics that can reduce unnecessary loop-unrollings and case-splittings. It is also interesting to see how these two automatic refinement approaches can be integrated to further speed up the convergence of our refinement loop into the right level of abstraction.

## ACKNOWLEDGMENT

The authors thank Bryant York for his support. This research is partially supported by Intel Equipment Grant #33768.

## REFERENCES

- [1] S. Hazelhurst and C.-J. Seger, "A simple theorem prover based on symbolic trajectory evaluation and OBDDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 4, pp. 413–422, April 1995.
- [2] C.-J. Seger and R. Bryant, "Formal verification by symbolic evaluation of partially-ordered trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–190, March 1995.
- [3] C.-T. Chou, "The mathematical foundation of symbolic trajectory evaluation," in *CAV'1999*, July 1999.
- [4] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. The MIT Press, 1999.
- [5] O. Coudert, J. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," in *Proc. of CAV'90*, 1990, pp. 23–32.
- [6] K. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
- [7] J. Yang and C.-J. Seger, "Generalized symbolic trajectory evaluation," *Intel SCL Technical Report*, 2000.
- [8] J. Yang and C.-J. H. Seger, "Introduction to generalized symbolic trajectory evaluation," *Transaction on VLSI Systems*, vol. 11, no. 3, June 2003.
- [9] J. Yang and C. J. H. Seger, "Generalized symbolic trajectory evaluation - abstraction in action," in *Proc. of FMCAD*, November 2002.
- [10] C.-J. Seger, R. Jones, J. O'Leary, T. Melham, M. Aagaard, C. Barrett, and D. Syme, "An industrially effective environment for formal hardware verification," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, 2005.
- [11] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
- [12] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, 2000, pp. 154–169.
- [13] R. Tzoref and O. Grumberg, "Automatic refinement and vacuity detection for symbolic trajectory evaluation," in *STE Symposium, CAV*, 2006, pp. 190–204.
- [14] J.-W. Roorda and K. Claessen, "Sat-based assistance to abstraction refinement for symbolic trajectory evaluation," in *Proc. of CAV*, 2006.

# A Logic for GSTE

Edward Smith

Oxford University Computing Laboratory

Wolfson Building, Parks Road, Oxford, OX1 3QD, England

Telephone: +44 1865 273838 Fax: +44 1865 273839

Email: ed.smith@comlab.ox.ac.uk

**Abstract**—The formal hardware verification technique of Generalized Symbolic Trajectory Evaluation (GSTE) traditionally uses diagrams called *assertion graphs* to express properties. Although the graphical nature of assertion graphs can be useful for understanding simple properties, it places limitations on formal reasoning. Clean reasoning is important for high-level verification steps, such as property decomposition. In GSTE, formal reasoning is also required to justify abstraction refinement steps, which are achieved via property transformation. This paper proposes a linear temporal logic, *generalized trajectory logic*, to provide a basis for reasoning about GSTE properties. The logic is textual and algebraic in nature, so it induces clean reasoning rules. We describe model checking for the logic, and look at rules for semantic equality, property decomposition and abstraction refinement.

## I. INTRODUCTION

*Generalized Symbolic Trajectory Evaluation* (GSTE) [1] is a form of model checking used for formal hardware verification. Developed by Intel as an extension to *Symbolic Trajectory Evaluation* (STE) [2], it has shown itself to be successful at verifying components of next-generation microprocessors [3]. GSTE uses operations based on partial gate-level simulation to check how a circuit reacts to particular patterns of input traces. It has several variants, including a fairness extension for  $\omega$ -regular properties. Some of these variants correspond to standard model checking algorithms [4]. In this paper, we consider the foundational *strong semantics* [1]. This is the most commonly used form in practice, and expresses regular properties. We will also make use of the compositional model checking extension from [5].

GSTE is fundamentally distinguished from other forms of model checking by the abstract representation that it uses for sets of circuit states. One key advantage of this representation is that each particular set is represented by a *range* of different approximations. The less precise a particular approximation is, the less space it consumes in the model checker. Therefore an abstraction trade-off can often be achieved between losing too much information and requiring too much space. User feedback is used to refine abstraction, by directing the model checker on how to pick the best representatives for verifying a given property. The structure of the property itself determines the control flow of model checking, which in turn determines the degree of abstraction employed.

Traditionally, specifications are expressed using the graphical notation of *assertion graphs* [1]. Assertion graphs are a variety of universal-automata [6], where each run represents

an assertion made about the allowable execution traces. The GSTE algorithm directly traverses the graph during model checking, so its shape directly affects the level of model checking abstraction. To control abstraction, manual transformations are applied directly to the graph.

The graphical nature of assertion graphs can be useful for displaying simple properties in a visual manner, but it places strict limitations on formal reasoning. Formal reasoning is important for managing verification, as experience from STE has shown [7]. Although some progress has been made on reasoning with assertion graphs [6], [8], the resulting rules tend to be complicated, because they deal directly with graphical structures. The compositional GSTE model checker further complicates matters by using specifications that mix assertion graphs with logical syntax. For cleaner reasoning, we require a unified property representation that is more structured and controlled.

In this paper we address this problem by introducing a linear-time temporal logic, *generalized trajectory logic* (GTL), for representing GSTE properties. We describe a model checking algorithm for GTL that simplifies compositional GSTE, whilst maintaining its core methodology. We believe that GTL is a good match for GSTE, and also induces the fundamental logical characteristics that are necessary for formal reasoning. We demonstrate this by building up rules for semantic equivalence, property decomposition and abstraction refinement, each illustrated with examples.

The only other existing language for specifying runs of GSTE is the language of *compositional specification* described in [5]. Although we draw inspiration from this language, there are several important differences between it and GTL. First and foremost, the language of compositional specification was primarily introduced to explain compositional GSTE model checking. Therefore it does not address other important aspects of GSTE, such as the symbolic features that play an important role in abstraction control. Secondly, we define our language as a temporal logic. This brings in associated results and intuition. Thirdly, we believe that our logic has a simpler semantics that makes it more amenable to reasoning and composition.

## II. GENERALIZED SYMBOLIC TRAJECTORY EVALUATION

This section contains an overview of GSTE, highlighting the important characteristics that govern the language choices that we have made in defining our logic.

### A. Model Checking Outline

GSTE properties make assertions about the state of a circuit after it has received a particular pattern of inputs. The pattern of inputs is called the *antecedent*, and the assertion is called the *consequent*. The *image* of an antecedent is the set of states that a circuit can be in after it has performed an execution trace that satisfies the antecedent. In GSTE, the antecedent drives a calculation of its own image, which is then compared with the consequent. Since it is based on forwards simulation, GSTE can only verify effect-of-cause properties, which is why properties are separated out into antecedent (cause) and consequent (effect) parts.

For example, suppose we wish to verify that a register correctly stores the number zero. We can use antecedent: “The register has most recently been written to with value zero, and is currently being read”. and consequent: “The output is zero”. Running GSTE effectively tests the consequent for every trace under which the antecedent holds.

In other GSTE formalisms, multiple antecedent/consequent assertions are combined into a single model checking run. This is so that a single simulation can take place with multiple consequents checks made along the way. For *reasoning* about properties, however, it is useful to keep the antecedents and consequents distinct, since this leads to a much simpler semantics. If necessary, properties can be combined into a single more efficient run after reasoning.

### B. Circuit Model

We model circuit states as boolean vectors in  $\mathbb{B}^{\mathcal{N}}$  where  $\mathcal{N}$  is the set of observable circuit *nodes*. We define  $S \subseteq \mathbb{B}^{\mathcal{N}}$  be the subset of *consistent* states that agree with the constraints imposed by the combinational logic of the circuit gates. A *circuit model* is a Kripke structure  $(S, T)$  where  $T \subseteq S \times S$  is a total model transition relation between consistent states. For the rest of the paper, we will assume that we are dealing with model  $M = (S, T)$ . We define  $\text{post}(R)$  to be the post-image of states  $R$  under  $T$ .

Circuit traces are modeled using non-empty finite words from  $S^+$ . We define  $\text{tr}(M)$  to be the set of such words that respect the model’s transition relation. We will write  $\text{last}(t)$  to mean the last state in trace  $t$ .

### C. Abstraction

Instead of storing precise sets of circuit states, GSTE uses a *ternary vector* representation to encode upper-approximations.

1) *Abstract State-Set Representation*: The ternary domain,  $\mathbb{T} = \{0, 1, X\}$ , is used to represent the possible values of a node in a particular set of states. Value 0 represents that a node is low in every state, 1 represents that a node is high in every state, and X that a node might have either value. We define the partial order  $\sqsubseteq_{\mathbb{T}}$  to provide a measure of how approximate these values are. Since X approximates 0 and 1, we have  $0 \sqsubseteq_{\mathbb{T}} X$  and  $1 \sqsubseteq_{\mathbb{T}} X$ .

In GSTE, sets of bit-vectors are represented using ternary vectors in  $\mathbb{T}^{\mathcal{N}}$ . For example, the set  $\{0011, 0001\}$  can be represented precisely using the ternary vector 00X1, and

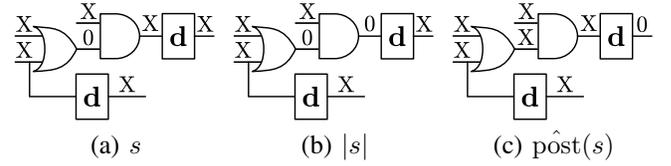


Fig. 1. Ternary simulation in GSTE

approximately by 0XX1, or even XXXX. Some sets cannot be represented precisely using ternary vectors alone. For example, the most precise ternary representation for  $\{0011, 0101\}$  is 0XX1. However, GSTE includes symbolic constructs that can be used to prevent this loss of precision, as we will see.

The use of ternary vectors is an example of a Cartesian abstraction [9], and *ignores dependencies* between circuit nodes. It is successful in hardware verification because useful properties often correspond to simple constraints on a small number of circuit nodes, which can be efficiently and precisely encoded as ternary vectors. Ternary vectors are also directly amenable to simulation-based calculations.

The relation between ternary vectors and sets of circuit states has been explained in [10] using the theory of abstract interpretation. By extending  $\sqsubseteq_{\mathbb{T}}$  point-wise to vectors and adding a bottom element, a complete lattice of ternary vectors is obtained. This relates to the complete lattice of sets of states, ordered by inclusion, via the concretization function  $\gamma : \mathbb{T}^{\mathcal{N}} \cup \{\perp\} \rightarrow 2^S$ , defined by

$$\gamma(a) = \begin{cases} \emptyset & \text{for } a = \perp \\ \{s \in S \mid \forall n \in \mathcal{N} : s_n \sqsubseteq_{\mathbb{T}} a_n\} & \text{for } a \in \mathbb{T}^{\mathcal{N}}. \end{cases}$$

2) *Abstract Operations*: An example GSTE simulation is illustrated in Fig. 1. In this figure, the **d**-blocks represent delay elements. Fig. 1 (a) shows a ternary vector  $s$  that represents all consistent circuit states where the node marked 0 is low. GSTE simulation *propagates* this constraint, based on simulation of the circuit logic, to deduce more state constraints downstream. This is shown in Fig. 1 (b) and formalized by the propagation operation, written  $|\cdot|$ , which is a lower closure on ternary vectors [11]. We will call the image of this closure the set of *ternary propagations*. GSTE’s abstract post-image function,  $\hat{\text{post}}$ , is calculated by first performing a propagation, and then setting every node to X, except for the delay element outputs, which take on their previous input values (Fig. 1 (a)-(c)). In this way, the new simulation state only contains constraints carried-over from state-holding elements. Notice that this simulation approach eliminates the need for a monolithic transition relation.

To approximate union and intersection operations on sets of states, GSTE uses the *join*,  $\sqcup$ , and *meet*,  $\sqcap$ , on the lattice of ternary propagations. For example,  $X10 \sqcup 111 = X1X$  and  $X1X \sqcap 0XX = 01X$ .

Given a map  $f$ , on sets of states, and its abstract counterpart  $f^\sharp$ , defined over ternary vectors, we say that  $f^\sharp$  is a *sound* approximation of  $f$  if  $f(\gamma(a)) \subseteq \gamma(f^\sharp(a))$  for any ternary vector  $a$ . It is furthermore *complete* if it precisely represents  $f$ , i.e.  $f(\gamma(a)) = \gamma(f^\sharp(a))$ .

Every GSTE operation is a sound representation of its set-based counterpart, and so maintains upper-approximations of the simulated state-sets. However, not all operations are complete, so some state information can be lost. Since the propagation step only goes forwards, it can lose information about the values of preceding nodes. Join is also not complete—the join of 101 and 011 is XX1, but the union of  $\gamma(101)$  and  $\gamma(011)$  is a strict subset of  $\gamma(\text{XX1})$ . In contrast, it is easy to show that meet is a complete representation of intersection.

#### D. Symbolic State-Set Representations

It is often necessary, during practical verification, to repeat a particular computation on a large number of different inputs. If the computation and domain have suitable structure, some of this effort can be shared. For example, suppose we wish to calculate  $f(x, g(y))$  for  $(x, y)$  in  $\{(1, 0), (2, 0), (3, 0)\}$ . Even though we are computing the result for three different inputs, we need only perform the sub-computation  $g$  once, namely to obtain  $g(0)$ . GSTE uses symbolic representations to take advantage of this observation during abstract circuit simulation.

Let  $V$  be a finite set of Boolean-valued variables and  $\mathcal{V} = V \rightarrow \mathbb{B}$  be the set of their possible valuations. We will use  $R^\mathcal{V}$  to model the set of *symbolic representations* for set  $R$ . These are functions from variable valuations to  $R$ , defining how the representation evaluates in each valuation. We will use the notation  $x\langle\nu\rangle$  to mean the evaluation of symbolic representation  $x$  in valuation  $\nu \in \mathcal{V}$ . For example,  $(u, u \wedge v)$  is a symbolic representation of bit-pairs where  $(u, u \wedge v)\langle[u \mapsto 0, v \mapsto 1]\rangle$  evaluates to  $(0, 0)$ .

GSTE uses *symbolic representations of ternary propagations* during model checking [2]. Multiple model checking runs can be shared by encoding them as different valuations of one single symbolic run. For such sharing to take place, properties must also be specified symbolically. The role of particular variables in these properties often corresponds directly with specific data values in the circuit design, so these symbolic values ‘flow’ independently through the circuit during simulation. For example, the following symbolic property uses a vector of variables,  $\mathbf{u}$ , to check that a register functions for *any* data value:

*Property 1:* “The register has most recently been written to with value  $\mathbf{u}$ , and the register is currently being read” *leads to* “The data output is  $\mathbf{u}$ ”.

### III. GENERALIZED TRAJECTORY LOGIC

In this section we introduce *generalized trajectory logic* (GTL), a linear-time temporal logic that allows GSTE runs to be expressed precisely, in a clean formal setting. We describe how circuit properties can be expressed using this logic, and examine some of its semantic characteristics.

#### A. Formal Definition

Let  $V$  be the finite set of Boolean-valued variables that we will use for symbolic representation. The syntax of GTL is

$f ::= \text{tt}$	True
$\text{ff}$	False
$\text{n}$	Node is high
$\neg\text{n}$	Node is low
$f \wedge f$	Conjunction
$f \vee f$	Disjunction
$\mathbf{Y}f$	Yesterday
$Z$	Fixed-point variable
$\mu Z.f$	Least fixed-point
$Q \rightarrow f \mid f$	Symbolic conditional
$f(u := Q)$	Symbolic substitution

Fig. 2. Syntax of GTL

$\ \text{tt}\ _\rho^\nu$	$= S^+$
$\ \text{ff}\ _\rho^\nu$	$= \emptyset$
$\ \text{n}\ _\rho^\nu$	$= \{t \in S^+ \mid (\text{last}(t))_n = 1\}$
$\ \neg\text{n}\ _\rho^\nu$	$= \{t \in S^+ \mid (\text{last}(t))_n = 0\}$
$\ f \wedge g\ _\rho^\nu$	$= \ f\ _\rho^\nu \cap \ g\ _\rho^\nu$
$\ f \vee g\ _\rho^\nu$	$= \ f\ _\rho^\nu \cup \ g\ _\rho^\nu$
$\ \mathbf{Y}f\ _\rho^\nu$	$= \{t.s \in S^+ \mid t \in \ f\ _\rho^\nu\}$
$\ Z\ _\rho^\nu$	$= \rho(Z)\langle\nu\rangle$
$\ \mu Z.f\ _\rho^\nu$	$= \bigcap \{T\langle\nu\rangle \mid \forall \nu \in \mathcal{V}. \ f\ _{\rho[Z \mapsto T]}^\nu \subseteq T\langle\nu\rangle\}$
$\ Q \rightarrow f \mid g\ _\rho^\nu$	$= \text{if } Q\langle\nu\rangle \text{ then } \ f\ _\rho^\nu \text{ else } \ g\ _\rho^\nu$
$\ f(u := Q)\ _\rho^\nu$	$= \ f\ _{\rho[u \mapsto Q]}^\nu$

Fig. 3. Semantics of GTL

defined in Fig. 2, where  $\text{n}$  is any circuit node,  $u$  is any variable in  $V$  and  $Q$  is any Boolean predicate over  $V$ . We make the requirement that within  $\mu Z.f$ , every occurrence of  $Z$  in  $f$  is bound by an occurrence of  $\mathbf{Y}$ .

The semantics of GTL is defined by giving the set of words from  $S^+$  that each formula satisfies in each symbolic valuation  $\nu \in \mathcal{V}$ . We write  $\|f\|_\rho^\nu$  for this, where  $\rho$  is a function that provides a context for fixed-point variables.

The semantics is defined in Fig. 3. Every trace satisfies truth,  $\text{tt}$ , and no trace satisfies falsity,  $\text{ff}$ . A trace satisfies the atomic proposition  $\text{n}$ , or  $\neg\text{n}$ , if node  $\text{n}$  is high, or low, respectively, in the *final* state of the trace. The connectives  $\wedge$  and  $\vee$  are defined like their counterparts in propositional logic. Unlike propositional logic, however, we do not allow negation of arbitrary formulas. This is to be in alignment with GSTE’s use of upper-approximations, for which no sensible interpretation of negation exists.

The only temporal operator is the *Yesterday* operator, written  $\mathbf{Y}$ . Intuitively,  $\mathbf{Y}f$  expresses that  $f$  held one time-step ago. A trace  $t$  satisfies  $\mathbf{Y}f$  if  $t$  with its last state removed satisfies  $f$ . Past time is preferred over future time, as it complements the forwards nature of GSTE. In particular, past time allows us to define compositional simulation—we can simulate  $\mathbf{Y}f$  by first simulating  $f$ , and then calculating its post-image.

We handle fixed-point recursion in the manner of the  $\mu$ -calculi. Such recursion is what differentiates GSTE from STE, and allow us to simulate over unbounded time-frames.

Intuitively, this can be seen as finite recursion. For example,  $\mu Z.f \vee \mathbf{Y}Z$  expresses that  $f$  has held at some point in the past. In this respect, GTL bears resemblance to the linear-time  $\mu$ -calculus [12].

Finally, GTL has constructs for describing symbolic representations. The *symbolic conditional*  $Q \rightarrow f | g$  is equivalent to  $f$  in valuations where  $Q$  holds, and  $g$  otherwise. This is an extension of the symbolic guard construct in STE's Trajectory Evaluation Logic [2]. For example, the formula  $u \rightarrow n | \neg n$  describes the symbolic traces ending in states where node  $n$  has value  $u$ . We will write ' $n$  is  $u$ ' as short-hand for this. The *symbolic substitution* construct allows us to explicitly change the current variable valuation context. This is useful during model checking because it allows us to 'look-up' the sets of states simulated in a particular valuation. For example,  $(\mathbf{Y}(n \text{ is } u))(u := T)$  symbolically simulates the circuit with variable  $u$ , then, afterwards, uses the symbolic result to find out what would have happened if  $u$  was true.

### B. Syntactic Sugar

We define the following symbolic quantification shorthands:

$$\begin{aligned} (\exists u : f) & \quad \text{for} \quad f(u := T) \vee f(u := F) \\ \text{and} \quad (\forall u : f) & \quad \text{for} \quad f(u := T) \wedge f(u := F). \end{aligned}$$

We also define some traditional linear-time temporal operators for common temporal patterns. *Previously*  $f$ , written  $\mathbf{P}f$ , asserts that  $f$  held at some point in the past:  $\mathbf{P}f := \mu Z.f \vee \mathbf{Y}Z$ . A second operator, *f Since g*, written  $f \mathbf{S} g$ , expresses that  $f$  holds at every point backward in time until some point where  $g$  holds:  $f \mathbf{S} g := \mu Z.g \vee (f \wedge \mathbf{Y}Z)$ . *Previously* and *Since* mirror LTL's *Finally* and *Until* operators.

### C. Semantic Characteristics

This section contains two useful theorems about GTL. Firstly, we define  $\mathcal{F}_{Z,f,\rho}(R)$  to be the semantic value of  $f$  in formula context  $\rho$  extended by mapping variable  $Z$  to  $R$ :

$$\mathcal{F}_{Z,f,\rho}(R) = \| f \|_{\rho[Z \mapsto R]}.$$

*Theorem 1 (Continuity)*: For any increasing chain  $X_i$  of symbolic trace sets,  $\bigcup_i \mathcal{F}_{Z,f,\rho}(X_i)$  equals  $\mathcal{F}_{Z,f,\rho}(\bigcup_i X_i)$ . This theorem, shown by structural induction over the syntax of  $f$ , says that GTL formulas are continuous with respect to their free recursion variables. By Tarski's Fixed-point Theorem, this shows that the limit of the *approximants* over  $\omega$ , defined by  $\mu^0 Z.f := \text{ff}$  and  $\mu^{n+1} Z.f := f[(\mu^n Z.f)/Z]$ , is the same as the GTL fixed-point:

$$\bigcup_{n \in \mathbb{N}} \| \mu^n Z.f \|_{\rho} = \| \mu Z.f \|_{\rho}.$$

This provides a basis for model checking fixed-points using iterative simulation. As a result of continuity, every GTL formula is also monotonic with respect to its free formula variables. This allows GSTE abstraction to proceed using upper-approximations. GTL also has unique fixed-points, which provide a useful reasoning device:

*Theorem 2*: If every occurrence of  $Z$  in  $f$  is bound by  $\mathbf{Y}$  then  $\mathcal{F}_{Z,f,\rho}$  has a unique fixed-point.

### D. GTL Properties

We will say that a GTL formula is closed if every occurrence of a recursion variable is bound by a corresponding  $\mu$ -expression. A GTL property is defined to be of the form antecedent  $A$  *leads to* consequent  $C$ , written  $A \rightarrow C$ , where  $A$  and  $C$  are both closed formulas of GTL. For example, we can express Property 1 from Section II-D as

$$\text{rd} \wedge ((\neg \text{wr}) \mathbf{S} (\text{wr} \wedge \text{in is } \mathbf{u})) \rightarrow \text{out is } \mathbf{u}.$$

The language satisfied by  $A \rightarrow C$ , consists of those words where  $A$  implies  $C$  under every symbolic valuation:

$$\mathcal{L}(A \rightarrow C) = \bigcap_{\nu \in \mathcal{V}} (S^+ \setminus \| A \|_{\nu}^{\vee}) \cup \| C \|_{\nu}^{\vee}.$$

We will say that model  $M$  satisfies property  $\mathcal{P}$ , written  $M \models \mathcal{P}$  when the traces of the model are included in this language,  $\text{tr}(M) \subseteq \mathcal{L}(\mathcal{P})$ .

### E. GTL Vector Properties

Because of their flat syntax,  $\mu$ -expressions are useful for expressing and reasoning about fixed-points. However, nested fixed-point expressions can quickly become incomprehensible to the human reader. In such cases, we can define properties using mutually dependent systems of equations. For example, we can specify Property 1 equivalently as

$$\begin{aligned} \text{Write} & = \text{wr} \wedge \text{in is } \mathbf{u} \\ \text{Written} & = \text{Write} \vee (\neg \text{wr} \wedge \mathbf{Y}(\text{Written})) \\ \text{rd} \wedge \text{Written} & \rightarrow \text{out is } \mathbf{u}. \end{aligned}$$

Such systems of equations are guaranteed to have a unique solution by Theorem 2, provided that each recursive cycle passes through  $\mathbf{Y}$ . This style of presentation also allows us to specify multiple consequent checks in a single run of GSTE.

## IV. MODEL CHECKING

We describe a model checking algorithm for GTL properties that is a generalization of compositional GSTE [5] to our less restrictive specification notation. We simulate those execution traces that satisfy the antecedent, and then assert that the consequent holds for all of them. For simplicity, we will only describe model checking for properties with *atemporal* consequents (containing no  $\mathbf{Y}$ ). Since we are only considering regular properties, this does not pose a restriction in practice.

### A. Set-Based Model Checking

First, we define model checking in terms of sets of states. Similar to the standard approach for CTL [13], the *simulation* of a formula is defined by structural recursion on the syntax of a formula. For symbolic valuation  $\nu$  and fixed-point context  $\tau$ , Fig. 4 defines the simulation of  $f$ , written  $[f]_{\tau}^{\nu}$ . We say that set-based model checking succeeds, written  $\mathbf{GSTE}_{\text{set}}(M, A \rightarrow C)$ , when  $[A]_{\nu}^{\nu} \subseteq [C]_{\nu}^{\nu}$  in every valuation  $\nu$ .

*Theorem 3*: If  $C$  is atemporal then  $\mathbf{GSTE}_{\text{set}}(M, A \rightarrow C)$  implies  $M \models A \rightarrow C$ .

$$\begin{aligned}
[\text{tt}]_\tau^\nu &= S \\
[\text{ff}]_\tau^\nu &= \emptyset \\
[\text{n}]_\tau^\nu &= \{s \in S \mid s_n = 1\} \\
[\neg\text{n}]_\tau^\nu &= \{s \in S \mid s_n = 0\} \\
[f \vee g]_\tau^\nu &= [f]_\tau^\nu \cup [g]_\tau^\nu \\
[f \wedge g]_\tau^\nu &= [f]_\tau^\nu \cap [g]_\tau^\nu \\
[\mathbf{Y}f]_\tau^\nu &= \text{post}([f]_\tau^\nu) \\
[\mu Z.f]_\tau^\nu &= \bigcup_{n \geq 0} [\mu^n Z.f]_\tau^\nu \\
[Q \rightarrow f \mid g]_\tau^\nu &= \text{if } Q(\nu) \text{ then } [f]_\tau^\nu \text{ else } [g]_\tau^\nu \\
[f(u := Q)]_\tau^\nu &= [f]_\tau^{\nu[u \mapsto Q(\nu)]}
\end{aligned}$$

Fig. 4. Set-based simulation

$$\begin{aligned}
[\text{tt}]_\sigma^\nu &= \lambda m. X \\
[\text{ff}]_\sigma^\nu &= \perp \\
[\text{n}]_\sigma^\nu &= \lambda m. \begin{cases} 1 & \text{if } n = m \\ X & \text{otherwise} \end{cases} \\
[\neg\text{n}]_\sigma^\nu &= \lambda m. \begin{cases} 0 & \text{if } n = m \\ X & \text{otherwise} \end{cases} \\
[f \vee g]_\sigma^\nu &= [f]_\sigma^\nu \sqcup [g]_\sigma^\nu \\
[f \wedge g]_\sigma^\nu &= [f]_\sigma^\nu \sqcap [g]_\sigma^\nu \\
[\mathbf{Y}f]_\sigma^\nu &= \text{p\hat{o}st}([f]_\sigma^\nu) \\
[\mu Z.f]_\sigma^\nu &= \bigsqcup_{n \geq 0} [\mu^n Z.f]_\sigma^\nu \\
[Q \rightarrow f \mid g]_\sigma^\nu &= \text{if } Q(\nu) \text{ then } [f]_\sigma^\nu \text{ else } [g]_\sigma^\nu \\
[f(u := Q)]_\sigma^\nu &= [f]_\sigma^{\nu[u \mapsto Q]}
\end{aligned}$$

Fig. 5. Abstract simulation

*Proof:* We define the image of formula  $f$  in valuation  $\nu$  to be  $\text{image}_M(f, \nu) = \text{last}(\text{tr}(M) \cap \|f\|^\nu)$ . It can be shown by structural induction that  $\text{image}_M(A, \nu) \subseteq [A]^\nu$  for any formula  $A$ , and  $\text{image}_M(C, \nu) = [C]^\nu$  for any atemporal formula  $C$ . Suppose  $t$  is a trace of  $M$  that satisfies  $A$ . Then  $\text{last}(t)$  is in the image of  $A$  by definition. If model checking succeeds, it must therefore also be in  $\text{image}_M(C, \nu)$ . Now, since  $C$  is atemporal, any trace ending in  $\text{last}(t)$  must also satisfy it. Hence  $t$  satisfies  $C$  and  $M \models A \rightarrow C$ . ■

Notice that the algorithm is not complete, because we are using a branching simulation algorithm to verify a linear logic. For example,  $\mathbf{Y}n \wedge \mathbf{Y}\neg n$  is  $\text{ff}$ , but its simulation is not necessarily empty, since there may exist two states with a common successor, that each satisfy  $n$  and  $\neg n$  respectively.

### B. Abstract Model Checking

We can adapt our set-based algorithm to use the abstract GSTE state-set representation described in Section II-C2. This results in *abstract simulation*, defined in Fig. 5. For simplicity, this figure presents ternary vectors as functions from nodes to ternary values. Since the abstract operations for disjunction and post-image are not complete, they cannot be used during consequent simulation—it is unsound to over-approximate the consequent assertion. Abstract model checking succeeds, written  $\text{GSTE}(M, A \rightarrow C)$ , when  $[A]^\nu \sqsubseteq [C]^\nu$  for every valuation  $\nu$ .

*Theorem 4:* If  $C$  is atemporal and does not contain disjunction then  $\text{GSTE}(M, A \rightarrow C)$  implies  $M \models A \rightarrow C$ .

*Proof:* Since every abstract GSTE operation is sound,  $[A]^\nu \subseteq \gamma([A]^\nu)$ . For atemporal consequents without disjunction, abstract simulation is furthermore complete:  $[C]^\nu = \gamma([C]^\nu)$ . Therefore  $[A]^\nu \sqsubseteq [C]^\nu$  implies  $[A]^\nu \subseteq [C]^\nu$  which in turn implies  $M \models A \rightarrow C$  by Theorem 3. ■

## V. EXPRESSIBILITY

Both the traditional specification notation of assertion graphs [1] and that of compositional specification [5] express the regular class of properties. This can be shown via the constructions to and from regular automata, as described in [4], [6].

For every assertion graph, we can directly construct an equivalent GTL vector property. Our construction creates a recursion variable  $X_e$  for each edge  $e$  in the assertion graph. This is used to capture the traces satisfied up to and including that edge, by any path from the initial vertex. To satisfy an initial edge, a trace must simply satisfy the atemporal antecedent on that edge,  $\text{ant}(e)$ . Thus we have  $X_e = \text{ant}(e)$ . To satisfy any other edge  $e$ , a trace must satisfy  $\text{ant}(e)$  and yesterday have satisfied some preceding edge of  $e$ . Therefore  $X_e$  equals the disjunction over every edge  $e'$  preceding  $e$ , of  $\text{ant}(e) \wedge \mathbf{Y}X_{e'}$ . For every consequent of the graph,  $C_e$  on edge  $e$ , we assert that  $X_e \rightarrow C_e$ . The resulting property specifies exactly the same behavior and model checking approach as the original assertion graph.

This construction effectively encodes the assertion graph GSTE algorithm explicitly into our logic. In the reverse direction, any GTL vector property with bounded consequents can also be re-written into an assertion graph form, via a possible exponential blow-up from expanding the product space that compositional GSTE reduces. Such GTL properties are therefore equally as expressive as both traditional assertion graphs and compositional GSTE specifications.

## VI. REASONING RULES

The main benefit of using GTL over assertion graphs is that its simple semantics provides a clean basis for reasoning. This section demonstrates this claim by describing a range of useful reasoning rules. It is important to realize that although some of these rules may appear simplistic, they often have no apparent counterpart for assertion graphs. They are therefore important contributions for dealing with properties.

### A. Equivalence Rules

First of all, we describe some fundamental rules for GTL formulas that preserve semantic equality. These are of practical use since they represent sound transformations of properties.

1) *Boolean Connectives:* Since  $\wedge, \vee, \text{tt}$  and  $\text{ff}$  follow the standard definitions of propositional logic, they have the same standard characteristics. The terms  $\text{tt}$  and  $\text{ff}$  behave as zeros and ones as expected, and there are the usual rules for commutativity and associativity, e.g.  $\text{tt} \wedge f = f$ ,  $f \wedge g = g \wedge f$  and  $f \wedge (g \wedge h) = (f \wedge g) \wedge h$ . Negation on atomic propositions behaves as expected:  $n \wedge \neg n = \text{ff}$ ,  $n \vee \neg n = \text{tt}$ .

2) *Fixed-points*: As we have already shown, fixed-points are unique and are the limit of their approximants from below. Fixed-points are also subject to the standard unrolling:  $\mu Z.f(Z) = f(\mu Z.f(Z))$ . From this it is easy to build up rules for common temporal patterns, such as  $\mathbf{P}f = f \vee \mathbf{Y}\mathbf{P}f$ . This might, for example, allow us to case-split based on whether  $f$  holds at the current point of time or not.

3) *Yesterday*: GTL is a linear logic, so Yesterday distributes over the other logical connectives. For example,  $\mathbf{Y}f \wedge \mathbf{Y}g = \mathbf{Y}(f \wedge g)$  and  $u \rightarrow \mathbf{Y}f \mid \mathbf{Y}g = \mathbf{Y}(u \rightarrow f \mid g)$ . Notice that  $\mathbf{Y}\mathbf{t}\mathbf{t} \neq \mathbf{t}\mathbf{t}$ , since  $\mathbf{Y}\mathbf{t}\mathbf{t}$  does not satisfy traces of length one. This may seem unusual, but it accurately represents our model of GSTE when not every state has a pre-image.

4) *Symbolic Constructs*: The symbolic nature of GTL is orthogonal to its other aspects. Therefore the logical operations distribute over symbolic ones. For example,  $(Q \rightarrow f \mid g) \wedge h = Q \rightarrow (g \wedge h) \mid (f \wedge h)$ ,  $(\mathbf{Y}f)(u := Q) = \mathbf{Y}(f(u := Q))$ . Rules for introducing and removing variables allow us to control whether property aspects are represented explicitly or symbolically. For example, the rule  $f \vee g = (\exists u : u \rightarrow f \mid g)$  allows us to change the two explicit post-image calculations  $\mathbf{Y}f \vee \mathbf{Y}g$  into the single symbolic post-image calculation given by  $(\exists u : \mathbf{Y}(u \rightarrow f \mid g))$ . The symbolic if-then-else also satisfies standard rules such as  $(\mathbf{T} \rightarrow f \mid g) = f$ ,  $(\mathbf{F} \rightarrow f \mid g) = g$  and  $(Q \rightarrow f \mid f) = f$ . Symbolic substitution is equivalent to textual substitution on its operand:  $f(u := Q) = f[Q/u]$ . This rule can be used to ‘flatten’ symbolic model checking into explicit model checking, as the following example demonstrates.

*Example*: Suppose we would like to verify that a 4-step clock generator signals `clk` when reset with signal `r`, and every fourth time interval afterwards. We can model check this behavior explicitly with the following property:

$$\begin{aligned} \text{Count}_0 &= r \vee (\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}\text{Count}_0)))) \\ \text{Count}_0 &\rightarrow \text{clk}. \end{aligned}$$

Model checking calculates the set of states where either a reset occurs, or has last occurred a multiple of four time-steps ago. Now suppose that we know that the timer is implemented using a two-bit counter, and we would like to use a more efficient symbolic model checking approach. The following property finds the symbolic set of states in which the value of the counter is `u`:

$$\begin{aligned} \text{Count} &= ((\mathbf{u} = 0) \rightarrow r \mid \text{ff}) \vee (\neg r \wedge \mathbf{Y}\text{Count}(\mathbf{u} := \mathbf{u} - 1)) \\ \text{Count}(\mathbf{u} := 0) &\rightarrow \text{clk}. \end{aligned}$$

It is not immediately obvious that the two properties are equivalent. However, using the substitution rule, we can deduce that

$$\begin{aligned} &\text{Count}(\mathbf{u} := 0) \\ &= \text{Count}[0/\mathbf{u}] \\ &= ((0 = 0) \rightarrow r \mid \text{ff}) \vee (\neg r \wedge \mathbf{Y}\text{Count}(\mathbf{u} := 0 - 1)) \\ &= r \vee (\neg r \wedge \mathbf{Y}\text{Count}[3/\mathbf{u}]) \\ &= r \vee (\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}\text{Count}[2/\mathbf{u}])) \\ &= r \vee (\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}\text{Count}[1/\mathbf{u}]))) \\ &= r \vee (\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}(\neg r \wedge \mathbf{Y}\text{Count}[0/\mathbf{u}]))))). \end{aligned}$$

Now we can use the unique fixed-point theorem to deduce that  $\text{Count}_0 = \text{Count}[0/\mathbf{u}]$ .

## B. Decomposition Rules

We will now consider rules that enable decomposition of a property into multiple model checking runs. Recall that the GTL property  $A \rightarrow C$  is satisfied if and only if every model trace that satisfies  $A$  also satisfies  $C$ . Therefore, for any circuit model, the leads-to relation is transitive:

$$\frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C}$$

This rule allows us to connect two GSTE simulations together. Since GTL formulas are also monotonic with respect to formula variables, transitivity induces generic substitution rules that can be used to piece together branching simulations:

$$\frac{A_1 \rightarrow C_1 \quad A_2 \rightarrow C_2}{A_1[A_2/C_2] \rightarrow C_1} \quad \frac{A_1 \rightarrow C_1 \quad A_2 \rightarrow C_2}{A_1 \rightarrow C_1[C_2/A_2]} \quad (1)$$

Properties can also be split based on the value of a symbolic variable. For example, if we can verify a property for both valuations of a particular variable, then the entire property must hold:

$$\frac{A(u := \mathbf{T}) \rightarrow C(u := \mathbf{T}) \quad A(u := \mathbf{F}) \rightarrow C(u := \mathbf{F})}{A \rightarrow C}$$

There are also more straight-forward rules for decomposition, such as antecedent and consequent splitting:

$$\frac{A_1 \rightarrow C \quad A_2 \rightarrow C}{A_1 \vee A_2 \rightarrow C} \quad \frac{A \rightarrow C_1 \quad A \rightarrow C_2}{A \rightarrow C_1 \wedge C_2}$$

We can also derive rules for higher-level temporal patterns. For example, we can verify that some invariant  $I$  holds perpetually after reset  $R$ , by first showing that the reset establishes the invariant, and then that the invariant inductively holds:

$$\frac{R \rightarrow I \quad \mathbf{Y}I \rightarrow I}{\mathbf{P}R \rightarrow I}$$

*Example*: To illustrate the use of one of these rules, we will examine the decomposition of the verification of an industrial memory design previously described in [6]. The design consists of two blocks: a memory block for storing incoming data, and a processing block that performs selection, alignment and mask on the data being read.

Verification aims to show that if data  $D$  has been written to address  $A$ , and not overwritten since, then if address  $A$  is accessed with appropriate options, the correct selection, alignment and mask of  $D$  is returned. The selection and alignment options must be provided with the read request, and the mask options provided one cycle later, when the read completes. Introducing extra names to describe the simple input predicates required, the property can be specified with GTL as:

$$\mathbf{Y}\mathbf{Y}(\text{no\_overwrite } \mathbf{S} \text{ write}) \wedge \mathbf{Y}(\text{read} \wedge \text{sel\_align}) \wedge \text{mask} \rightarrow \text{data\_correct}. \quad (2)$$

The decomposition of this property approximately halves verification time by introducing an internal predicate, `read_result`, to assert that the data is correctly transmitted on the bus between memory and processing blocks. The first stage of verification checks that the memory correctly stores the data and sends it on this bus:

$$\mathbf{Y}(\text{no\_overwrite } \mathbf{S} \text{ write}) \wedge \text{read} \rightarrow \text{read\_result}. \quad (3)$$

The second stage verifies that if the processing block correctly receives the data then it is processed correctly:

$$\mathbf{Y}(\text{read\_result} \wedge \text{sel\_align}) \wedge \text{mask} \rightarrow \text{data\_correct}. \quad (4)$$

To justify such decomposition it is necessary to show that (2) is implied by (3) and (4) together. Using only assertion graphs, it is difficult to establish this. In fact, the approach in [6] constructs a special monitor circuit and extra GSTE run to check this implication. However, by using GTL properties, the implication can easily be established with the application of the antecedent substitution rule in (1): (2) is obtained by substituting the antecedent of (3) for `read_result` in (4).

### C. Abstraction Refinement Rules

GTL is also effective for managing abstraction refinement steps. Recall that properties not only express *what* is being checked, but the shape of an antecedent formula also describes *how* to check it. Abstraction refinement corresponds to semantic-preserving re-writes that change the model checking direction. Since every atomic step of GSTE can be described precisely using GTL, we have complete control over model checking.

We will say that GTL formula  $f$  has an *abstraction refinement*  $g$ , written  $f \succsim g$ , if  $f$  and  $g$  are semantically equivalent and the abstract simulation of  $g$  is more precise than the abstract simulation of  $f$ , i.e.  $\lfloor f \rfloor_{\sigma}^{\nu} \sqsupseteq \lfloor g \rfloor_{\sigma}^{\nu}$ . Abstract simulation is monotonic with respect to  $\succsim$ , so abstraction rules can be soundly applied to any sub-formula of an antecedent:  $f \succsim g$  implies  $h[f/Z] \succsim h[g/Z]$ .

There are two ways in which GSTE simulation can over-approximate the image of an antecedent. Firstly, GSTE's abstract set representation introduces information loss due to its approximation of disjunction and post-image calculation. Secondly, as we have already discussed, set-based simulation itself is not complete for formulas that express what we term *product reduction*. We will consider rules for each of these effects in turn.

1) *Rules for Disjunction*: Suppose  $f$  does not contain any fixed-points, and model checking  $f(g \vee h) \rightarrow C$  fails due to over-abstraction. If the loss of required information is caused by this disjunction alone, then it must be that both  $f(g) \rightarrow C$  and  $f(h) \rightarrow C$  would succeed individually. We therefore have the option of simply *repeating* simulation  $f$  for both disjuncts independently. By doing this, we effectively make model checking more explicit. Such a refinement is captured by distributing  $f$  over disjunction:

$$f(g \vee h) \succsim f(g) \vee f(h).$$

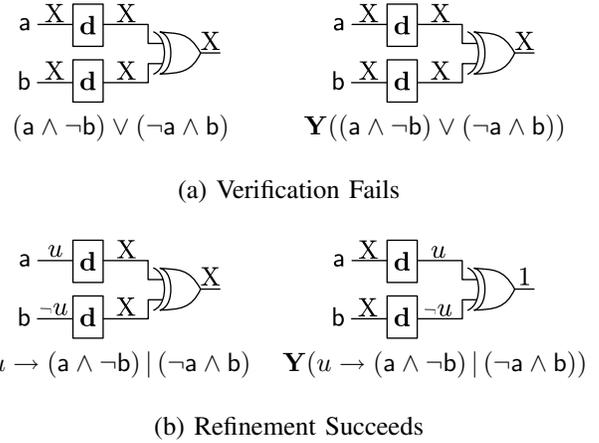


Fig. 6. Symbolic disjunctive completion

This refinement works because it delays the stage at which information is lost until later in the simulation. Since it increases the number of simulation steps that occur during model checking, it has a performance penalty. However, symbolic representation can allow us to *share* the common elements between two such repeated steps. We will term this technique *symbolic disjunctive completion* due to the correspondence with disjunctive completion in abstract interpretation theory [14]. Symbolic disjunctive completion uses  $u \rightarrow g | h$  to represent  $g \vee h$ , where  $u$  is existentially quantified at the top level of simulation. The symbolic states can then capture node dependencies that are otherwise lost:

$$f(g \vee h) \succsim \exists u : f(u \rightarrow g | h).$$

*Example*: Consider the XOR-gate with delayed inputs shown in Fig. 6. We would like to verify that if the inputs did not match in the preceding time step, then the output is high. The obvious simulation  $\mathbf{Y}((a \wedge \neg b) \vee (\neg a \wedge b))$ , shown in Fig. 6 (a), loses all information in the first time step. This is because we are specifying a dependency between circuit nodes that ternary vectors cannot capture. If we apply the rule for symbolic disjunctive completion, then the variable captures the required dependency, and the output can be demonstrated to be high, as in Fig. 6 (b).

2) *Rule for Post-Image Refinement*: GSTE does not propagate constraints backwards. This means that the post-image operation can lose important information. A common technique to counter this effect is to split the simulation based on a circuit node further back in the circuit. Such a split for node  $n$  is captured by the following rule:

$$f \succsim (n \wedge f) \vee (\neg n \wedge f).$$

*Example*: Any state that satisfies the constraints shown in Fig. 1 (b) must have a post-image where *both* delay-elements output low (the output of the OR-gate is low, so both inputs must also be low). However, the post-image in Fig. 1 (c) does not capture this constraint. If the verification is split in four,

based on the values of the OR-gate inputs, then this constraint is maintained. This rules out the impossible cases where the OR-gate has a high input but low output.

3) *Rule for Product Reduction:* A GTL simulation can be reduced by splitting it into two sub-simulations, whose results are then conjunctively composed. This has the effect of exploring a product property space by exploring each of its quotients independently in turn. This decomposition is an abstraction, since it ignores the *interaction* between the two quotient simulations. The essence of such transformations is captured by:

$$\mathbf{Y}f \wedge \mathbf{Y}g \lesssim \mathbf{Y}(f \wedge g).$$

When applied over fixed-points, this rule can be used to describe partial order reductions, as it allows interleaved events to be simulated independently.

*Example:* Consider verifying a three-stage pipeline that independently decrements its two binary inputs,  $a$  and  $b$ , then adds them together. We can try to simulate the two inputs independently, and combine the results at the final stage using:  $(\mathbf{Y}\mathbf{Y}(a \text{ is } u)) \wedge (\mathbf{Y}\mathbf{Y}(b \text{ is } v))$ . Suppose this run fails due to over-abstraction and we discover that, in fact, the addition starts at the second pipeline stage. We refine the simulation to:  $\mathbf{Y}(\mathbf{Y}(a \text{ is } u) \wedge \mathbf{Y}(b \text{ is } v))$ , or even  $\mathbf{Y}\mathbf{Y}(a \text{ is } u \wedge b \text{ is } v)$ . These simulations are more complex, but may now maintain enough information for verification to succeed.

## VII. CONCLUSION

This paper has presented a new temporal logic, *generalized trajectory logic*, for describing runs of GSTE in a clean formal setting. The logic is equally as expressive as GSTE assertion graphs under their strong semantics, but is more amenable to formal reasoning due to its textual and algebraic nature. We have demonstrated this by describing simple rules for important verification steps, that are otherwise lacking for assertion graphs.

As well as easing verification efforts, we hope that this method of specification opens the way for further formal analysis of GSTE. We have shown that abstraction refinement of GSTE simulations generally corresponds to simple distributive laws in their GTL descriptions. Might it be useful to automatically apply such rules, so that property and abstraction concerns can be separated? Can we use information about the circuit layout to guide their application? Our reasoning rules also allow quotients of the property to be easily transformed back-and-forth between symbolic and explicit representations. Which particular property aspects are better represented symbolically?

Although GTL provides a sound logical basis for specification, it still operates at a detailed level. It would be useful to use GTL to connect GSTE to formalisms with richer types and specification constructs. The logic itself might also be extended to handle other GSTE techniques such as backwards simulation, precise nodes [1] and liveness.

## ACKNOWLEDGMENT

The author is grateful to Tom Melham for his supervision, to Jin Yang and the SCL group at Intel for their support and early feedback, and to Sara Adams, Magnus Bjork, Carl Seger and the reviewers for their useful comments.

## REFERENCES

- [1] J. Yang and C.-J. H. Seger, "Generalized symbolic trajectory evaluation - abstraction in action," in *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, M. Aagaard and J. W. O'Leary, Eds., vol. 2517. Springer, 2002, pp. 70–87.
- [2] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *Proceedings of the 28th Design Automation Conference*, A. R. Newton, Ed. ACM Press, 1991, pp. 397–402.
- [3] T. Schubert, "High level formal verification of next-generation microprocessors," in *Proceedings of the 40th conference on design automation*, I. Getreu, L. Fix, and L. Lavagno, Eds. ACM Press, 2003, pp. 1–6.
- [4] R. Sebastiani, E. Singerman, S. Tonetta, and M. Y. Vardi, "Gste is partitioned model checking," in *Proceedings of the 16th International Conference on Computer Aided Verification (CAV) 2004*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 229–241.
- [5] J. Yang and C.-J. H. Seger, "Compositional specification and model checking in gste," in *16th International Conference in Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Springer, 2004, pp. 216–228.
- [6] J. C. Alan J. Hu and J. Yang, "Reasoning about gste assertion graphs," in *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, ser. Lecture Notes in Computer Science, D. Geist and E. Tronci, Eds. Springer, 2003, pp. 170–184.
- [7] S. Hazelhurst and C.-J.H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 4, pp. 413–422, 1995.
- [8] G. Yang, J. Yang, W. N. N. Hung, and X. Song, "Implication of assertion graphs in gste," in *Proceedings of the 2005 conference on Asia South Pacific design automation*, T.-A. Tang, Ed. ACM Press, 2005, pp. 1060–1063.
- [9] P. Cousot and R. Cousot, "Formal language, grammar and set-constraint-based program analysis by abstract interpretation," in *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*. New York, NY, USA: ACM Press, 1995, pp. 170–181.
- [10] C.-T. Chou, "The mathematical foundation of symbolic trajectory evaluation," in *Proc. 11th International Computer Aided Verification Conference*, ser. Lecture Notes in Computer Science, N. Halbwahs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 196–207. [Online]. Available: citeseer.nj.nec.com/chou99mathematical.html
- [11] J.-W. Roorda and K. Claessen, "Explaining symbolic trajectory evaluation by giving it a faithful semantics," in *Proceedings of the First International Computer Science Symposium in Russia, (CSR 2006)*, ser. Lecture Notes in Computer Science, D. Grigoriev, J. Harrison, and E. A. Hirsch, Eds., vol. 3967. Springer, 2006, pp. 555–566.
- [12] M. Y. Vardi, "A temporal fixpoint calculus," in *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1988, pp. 250–259.
- [13] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.
- [14] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Antonio, Texas: ACM Press, 1979, pp. 269–282.

# Automatic Abstraction in Symbolic Trajectory Evaluation

Sara Adams, Magnus Björk, Tom Melham  
Oxford University Computing Laboratory  
Oxford, OX1 3QD, England  
Email: {sara,magnus,melham}@comlab.ox.ac.uk

Carl-Johan Seger  
Strategic CAD Labs, Intel Corporation  
Hillsboro, OR 97124, USA  
Email: carl.seger@intel.com

**Abstract**—Symbolic trajectory evaluation (STE) is a model checking technology based on symbolic simulation over a lattice of abstract state sets. The STE algorithm operates over families of these abstractions encoded by Boolean formulas, enabling verification with many different abstraction cases in a single model-checking run. This provides a flexible way to achieve partitioned data abstraction. It is usually called ‘symbolic indexing’ and is widely used in memory verification, but has seen relatively limited adoption elsewhere, primarily because users typically have to create the right indexed family of abstractions manually. This work provides the first known algorithm that automatically computes these partitioned abstractions given a reference-model specification. Our experimental results show that this approach not only simplifies memory verification, but also enables handling completely different designs fully automatically.

## I. INTRODUCTION

Symbolic Trajectory Evaluation (STE) is a model checking technology based on symbolic simulation over a lattice of abstract state sets [1]. STE provides a combination of abstraction and algorithmic efficiency for verification of memories and datapath-dominated designs, and has tackled numerous difficult industrial verification problems [2]–[4].

In the abstraction lattice at the heart of STE, each circuit node is assigned a value in the set  $\{0, 1, X\}$ , with ‘X’ representing an unknown or ‘don’t care’ value. An assignment of such values to every circuit node is an abstraction of a set of Boolean circuit states. It is abstract in the sense that it ambiguously stands for any one of a family of Boolean state sets, one for each replacement of every X by 0 or 1. The collection of all such abstractions forms a lattice, ordered by the amount of information about node values.

The STE model-checking algorithm uses three-valued circuit simulation [5] to compute a reachable abstract state-set in this representation, comparing this to a specification written in a weak linear-time temporal logic. The algorithm is space-efficient because it operates over abstractions of sets of states; any parts of the circuit function not relevant to the specification get ‘abstracted away’ to X. Any correctness result verified in this abstract model transfers over to the real, Boolean model of circuit states. Formally, there is a Galois connection between the three-valued model and the Boolean model of states [6].

On top of the abstraction lattice, STE provides a layer of symbolic representation whereby whole families of abstractions may be checked simultaneously in one run of the model-

checker. The abstractions are indexed symbolically by Boolean variables, and formulas of Boolean logic are computed to represent the resulting families of reachable abstract state sets. This mechanism, sometimes called ‘symbolic indexing’, provides a flexible way to achieve partitioned data abstraction. A typical example is a memory verification, in which an  $n$ -element memory is verified with an indexed family of  $n$  abstractions, one for each address at which some target data might be located.

The abstractions in an indexed family can overlap in flexible, though not quite arbitrary, ways. This representation can also record interdependencies among node values, and so greatly increases the expressive power of specifications for STE. In implementations, the formulas in the symbolic layer may be manipulated using BDDs or other (usually semi-canonical) representations, and decided using SAT or BDDs.

This abstraction machinery is controlled by the way in which a user writes properties for model checking. By careful coding of the property, the user can guide the symbolic simulation done during model-checking through the right layers of the abstract state lattice to verify the property with contained complexity. A good illustration of success is the content-addressable memory verification done by Pandey and colleagues [2], in which a careful encoding of properties gives a logarithmic reduction in complexity.

Controlling abstraction manually in this way can be difficult, especially if there are assumptions about the operating environment of the verification. The property encoding required for abstraction is often non-obvious and tricky to devise. This paper describes an automatic approach to STE abstraction. We present an algorithm that computes an indexed family of abstractions from the specification to be verified. The abstraction scheme is then encoded in the guards of the verification property using the indexing transformation algorithm of Melham and Jones [7]. The result is an automatic abstraction method for STE, which requires little or no user guidance. We illustrate the effectiveness of our method with verifications of a memory, a CAM, and a simple scheduler.

## II. ABSTRACTION IN STE

Verification properties in STE are called *trajectory assertions* and have the form  $A \Rightarrow C$ , where  $A$  and  $C$  are formulas of a simple linear-time temporal logic. The intuition is that the

*antecedent* formula  $A$  describes some initial conditions of the circuit inputs and states, and the *consequent*  $C$  specifies the values expected on circuit nodes as a response. The atomic propositions in  $A$  and  $C$  take the form ‘ $P \rightarrow n$  is 0’ or ‘ $P \rightarrow n$  is 1’, where  $n$  is the name of a circuit node and the *guard*  $P$  is a formula of propositional logic. The guard determines when the proposition is asserted: if  $P$  is true, then the node  $n$  must have the value 0 (or 1 respectively); if  $P$  is false, then there is no such assertion and  $n$  can have any value—including, for abstraction efficiency, the don’t care value  $X$ . Antecedents and consequents are essentially just conjunctions of these atomic propositions, possibly modified by the next-time temporal operator  $N$ .

The guards in a trajectory assertion are all formulas of propositional logic over some set of variables; different guards can share variables, but not all the variables need appear in every guard. For each assignment of truth-values to these variables, the trajectory assertion collapses into a property verifiable by three-valued simulation, with  $X$ s on all circuit nodes not forced to 0 or 1 by the antecedent or the circuit itself. Sets of reachable states are approximated by abstractions that assign a value in  $\{0, 1, X\}$  to each circuit node. Given a trajectory assertion, STE simultaneously computes the family of all such three-valued simulations, one for each satisfying assignment to the variables in the guards.

Users control this partitioned abstraction mechanism by appropriate selection of guards. The idea can be illustrated by the following trivial example. Consider a unit-delay AND-gate with three input nodes  $a$ ,  $b$ , and  $c$  and output node  $o$ . A verification that does not exploit the abstraction lattice is achieved by running STE on this trajectory assertion:

$$\begin{aligned} \overline{t_1} \rightarrow a \text{ is } 0 \text{ and } t_1 \rightarrow a \text{ is } 1 \text{ and} \\ \overline{t_2} \rightarrow b \text{ is } 0 \text{ and } t_2 \rightarrow b \text{ is } 1 \text{ and} \\ \overline{t_3} \rightarrow c \text{ is } 0 \text{ and } t_3 \rightarrow c \text{ is } 1 \\ \Rightarrow \\ N(\overline{t_1} \vee \overline{t_2} \vee \overline{t_3} \rightarrow o \text{ is } 0 \text{ and } t_1 \wedge t_2 \wedge t_3 \rightarrow o \text{ is } 1) \end{aligned} \quad (1)$$

This is just symbolic Boolean simulation. The antecedent attaches a distinct, unconstrained, propositional variable to each input node. And the consequent asserts that the expected Boolean function of these variables appears on the output. Note that all Boolean variables in STE appear in the guards; the constants  $a$ ,  $b$ , and  $c$  are node names, not variables.

STE’s abstraction lattice lets us reduce the number of variables needed to verify this gate. The key observation is that if any one input is 0, then the output will be 0 regardless of the other inputs. We can exploit this with  $X$ s to introduce abstraction in the model-checking run. For the AND gate, there are four cases to check; we can enumerate or ‘index’ these with two variables, say  $x_1$  and  $x_2$ . We write the following:

$$\begin{aligned} \overline{x_1} \wedge \overline{x_2} \rightarrow a \text{ is } 0 \text{ and} \\ x_1 \wedge \overline{x_2} \rightarrow b \text{ is } 0 \text{ and} \\ \overline{x_1} \wedge x_2 \rightarrow c \text{ is } 0 \text{ and} \\ x_1 \wedge x_2 \rightarrow a \text{ is } 1 \text{ and } b \text{ is } 1 \text{ and } c \text{ is } 1 \\ \Rightarrow \\ N(\overline{x_1} \vee \overline{x_2} \rightarrow o \text{ is } 0 \text{ and } x_1 \wedge x_2 \rightarrow o \text{ is } 1) \end{aligned} \quad (2)$$

Model-checking this with STE will simultaneously check four cases, each with different but sometimes overlapping abstractions of the reachable states arising. Any property verified in STE with a node set to  $X$  also holds when the node is either 0 or 1, so this assertion covers all input cases and is complete.

The advantage of this kind of abstraction is that it makes the representation of sets of states more compact, so that BDD or SAT computations in the model-checking are more tractable. The reduction in the number of propositional variables can be substantial in real applications.

### A. Indexing Transformations

Melham and Jones [7] describe an algorithm by which trajectory assertions can be transformed to introduce more abstraction. Suppose  $A \Rightarrow C$  is an assertion. The algorithm replaces the guards in  $A$  and  $C$  with new propositional formulas over a set of fresh variables in such a way that if the transformed assertion holds then so does  $A \Rightarrow C$ .

The algorithm takes as input a relation that specifies the abstraction scheme to be applied. For the three-input AND-gate, one possible abstraction relation is

$$\begin{aligned} ((\overline{x_1} \wedge \overline{x_2}) \rightarrow \overline{t_1}) \wedge ((x_1 \wedge \overline{x_2}) \rightarrow \overline{t_2}) \wedge \\ ((\overline{x_1} \wedge x_2) \rightarrow \overline{t_3}) \wedge ((x_1 \wedge x_2) \rightarrow (t_1 \wedge t_2 \wedge t_3)) \end{aligned} \quad (3)$$

The variables  $x_1$  and  $x_2$  index the abstraction cases, and the variables  $t_1$ ,  $t_2$ , and  $t_3$  appear in the directly-formulated Boolean trajectory assertion (1). Using this relation, the algorithm will compute the encoded trajectory assertion (2).

The Melham-Jones algorithm works by taking certain preimages of an assertion’s guards under the supplied abstraction relation. The relation takes the form  $R[\mathcal{X}, \mathcal{T}]$ , where  $\mathcal{T}$  is a set of *target variables* that occur in the guards of  $A \Rightarrow C$  (they need not be all the variables) and  $\mathcal{X}$  is a set of fresh *indexing variables*. For a given guard  $P$ , we define the *weak preimage*  $P_R$  and *strong preimage*  $P^R$  by:

$$\begin{aligned} P_R[\mathcal{X}] &= \exists \mathcal{T}. R[\mathcal{X}, \mathcal{T}] \wedge P[\mathcal{T}] \\ P^R[\mathcal{X}] &= P_R[\mathcal{X}] \wedge \neg \exists \mathcal{T}. R[\mathcal{X}, \mathcal{T}] \wedge \neg P[\mathcal{T}] \end{aligned}$$

where  $\exists \mathcal{T}$  denotes existential quantification over all the variables in the set  $\mathcal{T}$ . Intuitively,  $P_R[\mathcal{X}]$  is true for all indices  $\mathcal{X}$  that *allow*  $P$  to hold, and  $P^R[\mathcal{X}]$  is true for all indices  $\mathcal{X}$  that *force*  $P$  to hold. Given an assertion  $A \Rightarrow C$ , the algorithm applies the strong preimage to the guards in  $A$  and the weak preimage to the guards in  $C$ . This weakens the verification assumptions  $A$  by introducing  $X$ s, while maintaining the strength of the verification requirements  $C$ .

It is a technical side-condition required for soundness of the abstraction that the supplied relation satisfies the coverage condition  $\forall \mathcal{T}. \exists \mathcal{X}. R[\mathcal{X}, \mathcal{T}]$ . This ensures all the target variable values are indexed. Coverage is a bit more tricky when there are environmental constraints on the verification [7].

The indexing algorithm can be used on specifications that specify timing delays, such as the AND-gate above, as well as purely combinational ones. There is no explicit representation of time in the indexing relation itself—sequential STE specifications normally use distinct variables in the guards of

input nodes for the different time points of interest, so these are just different target variables in the indexing relation.

### B. The Automatic Abstraction Method

The core of the method in this paper is an algorithm that automatically computes abstractions for use with the indexing transformation just described. The output of the algorithm is an abstraction relation of the kind illustrated by (3), the AND-gate relation of the previous section, but vastly more complex and unintuitive for realistic examples. The practical benefit is that a user does not have to invent the abstraction and manually encode it into the trajectory assertion to be verified.

The algorithm takes as input a *specification* for the circuit to be verified, in the form of a Boolean expression that states the required I/O function. This specification is typically a component of the STE assertion to be verified, so in principle our method need not require anything beyond the manually-written properties that any verification needs. (But see the discussion of ‘symbolic constants’ below.) We work from specifications rather than circuits because they give a clean reference model of the algorithm the circuit uses, unencumbered by implementation detail [8]. Although the I/O specification provided as input to our algorithm is purely combinational, the resulting abstraction relation is applicable to sequential circuits, for the reasons explained in the previous section.

The algorithm is presented in Sections III and IV, where we also prove that the relations generated meet the required coverage condition by construction. In Section V we then describe several optimizations to the preimage calculations done to transform trajectory assertions. Some of these exploit the special form that our abstractions relations have because of how they are generated. We prove the correctness of the most complex of these optimizations.

The experimental results in Section VI show that our method handles both embedded memories—the classic target for manual symbolic indexing—and the much less intuitive example of a scheduler. The paper concludes with some discussion of related work and our plans for future extensions.

## III. COMPUTING ABSTRACTION RELATIONS

The approach we take in computing an abstraction relation is to use the structure of a specification function. Specifically, we assume that we have been given a Boolean expression constructed using only two-input AND operators, NOT operators, and named variables. We call this datatype a *bexpr* for short. For the time being, we suppose the expression has a single output and is tree-structured.

We introduce the algorithm by describing a simplified version, so we can convey the basic idea without too much detail. Consider the algorithm in Fig. 1. The function *simple\_bp* takes a *bexpr* for the specification and two BDDs, *h* and *l*, that represent the conditions under which the output should be high or low, respectively. It computes a BDD representing an abstraction relation by propagating conditions backwards through the circuit. The index variables of this relation are

```

1. simple_bp(e, h, l) =
2. if is_VAR(e) then
3.   t := bexpr2bdd(e)
4.   return((h → t) ∧ (l →  $\bar{t}$ ));
5. elseif is_NOT(e) then
6.   return bp(strip_NOT(e), l, h);
7. else // AND
8.   x := fresh_index_var();
9.   (e1, e2) := destruct_AND(e)
10.  r1 := simple_bp(e1, h, l ∧ x);
11.  r2 := simple_bp(e2, h, l ∧  $\bar{x}$ );
12.  return(r1 ∧ r2);

```

Fig. 1. Simple back-propagation algorithm.

generated in line 8 and occur in *h* and *l*. The target variables originate from the *bexpr* and are introduced in line 3.

The algorithm is recursive over the structure of *e*, the specification *bexpr*. In the base case, when the *e* is a variable, the abstraction relation simply says that the target variable is true whenever *h* holds and false whenever *l* holds. An invariant of the algorithm is that *h* and *l* never both hold. For NOT operations, we simply reverse *h* and *l* and continue. Finally, for a two-input AND operator, more work is needed. Both inputs must be high whenever *h* holds, so we simply pass *h* to the recursive calls in lines 10 and 11. But there are two ways *l* can be forced: either the first input or the second input has to be low. This choice is captured by creating a fresh index variable *x*. It is used to select which of the inputs are going to be the decisive low signal. Note that we never require both inputs to be low in order to achieve a maximum abstraction.

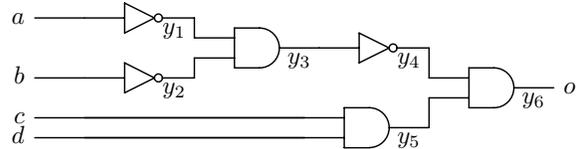


Fig. 2. Example circuit to illustrate our algorithm.

To illustrate the algorithm, consider the circuit in Fig. 2. First assume we want to compute the abstraction relation for the case the output is high. In other words, we want to compute a family of abstract inputs that all yield an output of high and that cover all such inputs. The resulting abstraction relation is

$$(x_1 \rightarrow a) \wedge (\bar{x}_1 \rightarrow b) \wedge c \wedge d.$$

Intuitively, the index variable *x*<sub>1</sub> decides whether *a* or *b* has to be high, and both *c* and *d* must be high. Similarly, if we call *simple\_bp* with *h*=F and *l*=T we get the abstraction relation

$$(x_1 \rightarrow \bar{a}) \wedge (x_1 \rightarrow \bar{b}) \wedge (x_2 \bar{x}_1 \rightarrow \bar{c}) \wedge (\bar{x}_2 \bar{x}_1 \rightarrow \bar{d}).$$

We can combine the above two cases by calling *simple\_bp* with *h*=*x*<sub>0</sub> and *l*= $\bar{x}_0$ , where *x*<sub>0</sub> is a fresh Boolean indexing

variable. In this combined case, we get

$$(x_3 x_0 \rightarrow a) \wedge (x_1 \bar{x}_0 \rightarrow \bar{a}) \wedge (\bar{x}_3 x_0 \rightarrow b) \wedge (x_1 \bar{x}_0 \rightarrow \bar{b}) \wedge (x_0 \rightarrow c) \wedge (x_2 \bar{x}_1 \bar{x}_0 \rightarrow \bar{c}) \wedge (x_0 \rightarrow d) \wedge (\bar{x}_2 \bar{x}_1 \bar{x}_0 \rightarrow \bar{d}).$$

There are several shortcomings of this simple algorithm that make it inefficient. First, it builds a monolithic Boolean expression for the abstraction relation. But a partitioned relation is often more feasible and easier to use. Second, the algorithm is very generous in using fresh Boolean variables. Third, by recognizing only two-input AND gates, the algorithm will essentially use a unary encoding, rather than a binary encoding. And by recognizing only conjunctions, the algorithm uses more Boolean variables than needed—for example, simply recognizing XNOR gates can reduce the number of variables by a factor of two. Finally, the algorithm is sometimes too aggressive in computing the abstraction relation. It is quite common that the user can select some signals not to be part of the abstraction. Thus, the algorithm needs to be controlled when such *symbolic constants* are present.

#### IV. IMPROVED ALGORITHM

In Fig. 3, we provide a much improved algorithm that tackles all of the shortcomings of the simple algorithm.

```

1.  $bp(\mathcal{C}, e, h, l, name) =$ 
2. if  $freevars(e) \subseteq \mathcal{C}$  OR  $is\_VAR(e)$  then
3.   return  $\{(bexpr2bdd(e), h, l)\};$ 
4. elseif  $is\_XNOR(e)$  then
5.    $(i_1, i_2) := sort\_inp\_args(\mathcal{C}, e);$ 
6.   if  $freevars(i_1) \subseteq \mathcal{C}$  then
7.      $c := bexpr2bdd(i_1);$ 
8.     return  $bp(\mathcal{C}, i_2, hc \vee l\bar{c}, h\bar{c} \vee lc, name);$ 
9.   else
10.     $\{x_1, x_2\} := get\_case\_exprs(name, 2);$ 
11.     $\{n_1, n_2\} := make\_unique\_names(name, 2);$ 
12.    return  $(bp(\mathcal{C}, i_1, hx_1 \vee lx_1, hx_2 \vee lx_2, n_1) \cup$ 
13.       $bp(\mathcal{C}, i_2, hx_1 \vee lx_2, hx_2 \vee lx_1, n_2) );$ 
14. elseif  $is\_NOT(e)$  then
15.   return  $bp(\mathcal{C}, strip\_NOT(e), l, h, name);$ 
16. else // AND
17.    $(cis, oinps) := find\_big\_ands(e);$ 
18.    $c := \bigwedge_{c_i \in cis} bexpr2bdd(c_i);$ 
19.    $res :=$  if  $cis = \emptyset$  then  $\emptyset$  else  $\{(c, h, F)\};$ 
20.    $cases := get\_case\_exprs(name, |oinps|);$ 
21.   if  $h \equiv F$  then
22.      $names := mk\_same\_names(name, |oinps|);$ 
23.   else
24.      $names := mk\_unique\_names(name, |oinps|);$ 
25.   foreach  $b \in oinps, s \in cases, n \in names$ 
26.      $res := res \cup bp(\mathcal{C}, b, h, l \wedge s \wedge c, n);$ 
27.   return  $res;$ 

```

Fig. 3. Main back-propagation algorithm.

The algorithm is recursive and takes a set of symbolic constants  $\mathcal{C}$ , a bexpr  $e$ , the high and low conditions  $h$  and  $l$ , and a base name  $name$  used to create unique variables. It returns a list of triples, where each triple consists of a Boolean function  $f$  over the symbolic constants  $\mathcal{C}$  and the generated indexing variables  $\mathcal{X}$ , and two Boolean functions denoting the cases in which  $f$  should be high or low, respectively. In Section V we will show how this format leads to a very efficient algorithm for computing the preimage operations needed for STE.

In more detail, on line 2-3, we deal with the two cases that the expression  $e$  only depends on variables in  $\mathcal{C}$  or  $e$  is a target variable. For either case, we create a singleton triple representing the partial relation.

On line 4, we test whether the last bexpr represents an XNOR gate or not. If yes, on line 5 we sort the two inputs to the XNOR gate by their support so that if one input only depends on variables in  $\mathcal{C}$ , it will be the first input  $i_1$ . If at least one of the inputs depends only on variables in  $\mathcal{C}$ , we do not need to introduce any new indexing variables, but can simply call  $bp$  recursively suitably modifying the  $h$  and  $l$  functions given the value of the symbolic expression. If both inputs to the XNOR depend on target variables, then on line 10-11 we create two case expressions and two new base names. The case expressions are simply  $x_i$  and  $\bar{x}_i$  for some variable  $x_i$  created from the  $name$  argument. On lines 12 and 13 we then call  $bp$  recursively on inputs  $i_1$  and  $i_2$ , with suitable high and low functions and distinct base names. Finally, the union of the two sets of triples returned is formed.

On lines 14-15 we deal with the case when  $e$  is a NOT expression. Here we simply call  $bp$  recursively switching  $h$  and  $l$ .

Finally on lines 16-27 we deal with the case when the final operator of  $e$  is an AND gate. On line 17, we traverse the bexpr to find as large an AND gate as possible by calling the routine `find_big_ands`. This routine not only finds all inputs that are conjoined, it also separates the inputs into two groups: the ones that only depend on variables in  $\mathcal{C}$  and those that do not. The conjunction of the first group is formed in line 18, since they do not need indexing variables. Only a consistency requirement, computed on line 19, is needed to make sure the assignment of the indexing variables does not contradict the value of this expression. Technically, we are only required to add the implication  $h \rightarrow c$ , but rather than having a separate set of consistency requirements, we include the triple  $(c, h, F)$  in the relation.

On line 20 we compute a set of mutually exclusive Boolean expressions over some fresh indexing variables that will be used to encode which of the inputs should be set to low. On line 21, we deal with a common special case that allows us to call  $bp$  recursively without new base names, thus greatly increase the sharing of indexing variables. Finally, on lines 25 and 26 we call  $bp$  recursively on every input expression modifying the low condition according to the case expressions.

### A. Coverage

Although the intuition behind the algorithm is fairly simple, it is critical to ensure that the trajectory assertions after abstraction verify all of the cases that were verified by the original assertions. We guarantee this by proving coverage in Theorem 1. Note that apart from indexing variables  $\mathcal{X}$  we also need to consider the symbolic constants  $\mathcal{C}$ . Recall that  $\mathcal{X}$  are used to index different cases of assertions for  $\mathcal{T}$ , while the symbolic constants  $\mathcal{C}$  basically introduce some additional, temporary target variables. For coverage we hence show that for every assignment of target variables and symbolic constants we can find an indexing corresponding to it.

*Theorem 1 (Coverage):* Given a Boolean expression represented as a bexpr-tree  $e$ , let  $rl = bp(\mathcal{C}, e, x_0, \bar{x}_0, name)$  be the result of the algorithm in Fig. 3, where  $name$  is chosen to ensure that none of the generated variables are called  $x_0$ . If

$$R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = \bigwedge_{(e_i, h_i, l_i) \in rl} (h_i \rightarrow e_i) \wedge (l_i \rightarrow \bar{e}_i),$$

then  $\forall \mathcal{T}. \forall \mathcal{C}. \exists \mathcal{X}. R[\mathcal{X}, \mathcal{C}, \mathcal{T}]$ .

*Proof:* Let the target variables  $\mathcal{T}$  and the symbolic constants  $\mathcal{C}$  be arbitrary, but fixed. Then we can simulate the bexpr-tree. That is, we can determine all input and output values of each constructor of  $e$ . In particular, the simulation result of  $e$  and the values of all consequents  $e_n$  in the relation  $R[\mathcal{X}, \mathcal{C}, \mathcal{T}]$  are known. By induction on the depth  $k$  of the bexpr-tree  $e$  we show that there exists a variable assignment for  $\mathcal{X}$  such that

$$\forall n. ((e_n = F) \rightarrow (h_n = F)) \wedge ((e_n = T) \rightarrow (l_n = F)) \quad (4)$$

Then  $R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = T$  as required.

*Base case:* The only bexpr-tree  $e$  of depth 0 is a single variable  $t$ . In this case the relation is  $R[\mathcal{X}, \mathcal{C}, \mathcal{T}] = (x_0 \rightarrow t) \wedge (\bar{x}_0 \rightarrow \bar{t})$ . Property (4) is satisfied by choosing  $x_0 = t$ .

*Inductive step:* Suppose that for all bexpr-trees of depth  $\leq k$  there is a variable assignment such that (4) holds.

We examine the last constructor of the bexpr-tree  $e$  of depth  $k + 1$  and determine an assignment for the variables  $\mathcal{Y}$  introduced by the algorithm for that constructor. W.l.o.g. assume the last constructor is an AND or a NOT that completes an XNOR. In both cases assume at least one input depends on a variable not in  $\mathcal{C}$ —otherwise, no new indexing variables are introduced and the induction hypothesis provides the required variable assignment for  $\mathcal{X}$ . We consider the two constructor cases, AND and NOT, in I and II below.

I. Suppose the last constructor is AND. We analyse  $e$  further to extract all consecutive AND constructors, resulting in a multiple input AND of maximal size, which is achieved by *find\_big\_and*s in line 17 of Fig. 3. We will now determine a variable assignment as desired depending on the simulation result of  $e$ . For this we split the inputs of the AND gate into two groups:  $cis$ , which only depend on variables in  $\mathcal{C}$ , and  $i_j$ , which depend on some indexing variables.

Ia. First suppose  $cis = \emptyset$ . Further suppose the simulation result of  $e$  is true. We choose  $x_0 = T$ , and hence

the output conditions are  $h_e = T, l_e = F$ . Observe that all inputs  $i_j$  of the AND have to be true in the simulation and  $\forall j. h_{i_j} = T, l_{i_j} = F$ . Arbitrary values can be chosen for  $\mathcal{Y}$ , they are only necessary for the case where  $e$  is false. Note that the variables  $\mathcal{Y}$  are never used at any other point in the algorithm due to the use of new, unique base names. An assignment for the remaining  $\mathcal{X}$  is determined as follows: By induction hypothesis there exists an assignment for the bexpr-tree of each input  $i_j$ . The algorithm guarantees that each input-tree uses a unique set of variables by specifying a unique base name for the variables. Hence the assignments for each input tree, and our assignment for  $\mathcal{Y}$ , can be merged to an assignment as required.

Ib. Now suppose  $cis = \emptyset$ , but the simulation result of  $e$  is false. We choose  $x_0 = F$ , and hence  $h_e = F, l_e = T$ . Observe that at least one input  $i_j$  of the AND has to be false in the simulation, w.l.o.g.  $i_1 = F$ . The algorithm specifies that  $h_{i_j} = l_e = F$  and  $l_{i_j} = case_j[\mathcal{Y}] \wedge l_e$ . We choose a variable assignment for  $\mathcal{Y}$  such that  $case_1 = T$  and hence  $case_j = F \quad \forall j \neq 1$ .

By induction hypothesis there exists a variable assignment for the input tree of  $i_1$  as required. It does not include  $\mathcal{Y}$ , so the variable assignment can be merged with our assignment to  $\mathcal{Y}$ . Observe that for all other inputs  $h_{i_j} = l_{i_j} = F, j \neq 1$ . But the algorithm forms new  $h_{in}, l_{in}$  using conjunction with  $h_{out}, l_{out}$  only, so we can deduce that all  $h, l$  on the input trees of  $i_j, j \neq 1$  will be false throughout, in particular at the terminating points  $e_n$  included in the relation. This means that all variable assignments, and in particular the one chosen for  $i_1$ , satisfy (4) trivially.

Ic. Next suppose  $cis \neq \emptyset$  and  $c = \bigwedge_{c_i \in cis} c_i$  is false. Then at least one of the concrete inputs is false, and accordingly the simulation output is false. Hence we choose  $x_0 = F$  and receive  $h_e = F$ . The extra requirement  $h_e \rightarrow c$  is therefore trivially satisfied. Also notice that in this case we need not force any other inputs to be low, i.e.  $l_{i_j} = F$  for all  $j$ . So the indexing variables  $\mathcal{Y}$  can be chosen arbitrarily with the same reasoning as in Ia.

Id. Finally suppose  $cis \neq \emptyset$  and  $c = \bigwedge_{c_i \in cis} c_i$  is true. Then the extra requirement  $h_e \rightarrow c$  is trivially satisfied. If the simulation output is true, we choose  $x_0 = T$  and  $\mathcal{Y}$  can be chosen arbitrarily as seen in Ia. If the simulation output is false, we choose  $x_0 = F$ . Given that we know that the concrete inputs are all true this means the indexing has to force one of the remaining inputs to be false. As seen in Ib, w.l.o.g. we choose  $\mathcal{Y}$  such that  $case_1 = T$  and use the induction hypothesis to complete the variable assignment.

II. Now suppose the last constructor is a NOT that completes an XNOR, where  $i_1$  and  $i_2$  are the two subtrees of  $e$  being compared. This is detected by *is\_XNOR* in line 2 of Fig. 3. Further assume that neither input only depends on  $\mathcal{C}$ . Recall that for XNORs the algorithm sets the input conditions  $h_{i_1} = y \wedge h_e \vee y \wedge l_e$  and  $l_{i_1} = \bar{y} \wedge h_e \vee \bar{y} \wedge l_e$  and  $h_{i_2} = y \wedge h_e \vee \bar{y} \wedge l_e$  and  $l_{i_2} = \bar{y} \wedge h_e \vee y \wedge l_e$  using the output conditions and a single indexing variable  $y$  solely introduced for that step. Intuitively, the indexing variables  $x_0$  and  $y$  enumerate the four possible

cases that can occur for the 2-bit XNOR. We now specify which assignments correspond to the explicit instances.

Suppose the simulation result shows that  $i_1 = F, i_2 = F, e = T$ . We choose  $x_0 = T$ , and hence  $h_e = T, l_e = F$ . By further choosing  $y = F$  we receive  $h_{i_1} = h_{i_2} = F$  and  $l_{i_1} = l_{i_2} = T$ , which corresponds to the input values. By induction hypothesis there exists an assignment for the bexpr-tree  $i_1$  and  $i_2$ . These two assignments and our assignment to  $x_0$  and  $y$  are merged and results in the full assignment required. Notice that  $i_1$  and  $i_2$  use distinct base name for their variables and never  $x_0$  or  $y$ , so no conflicts can occur.

The other three possible cases are similar. If simulation result shows that  $i_1 = T, i_2 = T, e = T$ , we choose  $x_0 = T$  and  $y = T$ . If it is  $i_1 = F, i_2 = T, e = F$ , we choose  $x_0 = F$  and  $y = F$ . Finally, if  $i_1 = T, i_2 = F, e = F$ , then we choose  $x_0 = F$  and  $y = T$ .  $\square$

### B. DAGs and multiple outputs

In the previous we assumed that we are given a bexpr-tree. The algorithm can be extended to handle DAGs as follows. Every fanout point is marked with a new variable  $v$  and the graph is cut at these points. The algorithm is then run for each resulting tree with distinct variables to guarantee independence of indexing variables. Observe that in some cases  $v$  will be an output, and in others an initial input. The resulting triplets are merged as follows.

Assume  $T_o$  is a set of triplets computed by the algorithm, where the output was previously a fanout point  $v$ . W.l.o.g. let  $v_h$  and  $v_l$  be two fresh, distinct variables used for  $h$  and  $l$  respectively when running the algorithm.

Further assume that  $T_i$  is a set of triplets computed where  $v$  is one of the initial inputs, i.e.  $v$  is a target variable. Let  $(v, h_v, l_v) \in T_i$  be the triplet corresponding to  $v$ .

We then determine  $T'_o$  by substituting  $v_h$  with  $h_v$  and  $v_l$  with  $l_v$  in every triplet of  $T_o$ . This disposes of the variables  $v_h$  and  $v_l$ . Additionally, due to the fact that  $h_v$  and  $l_v$  are mutex, contradicting assignments for  $v_h$  and  $v_l$  are eliminated. The union of all  $T_i$  and  $T'_o$  for all cut points then represents the set of all triplets needed for the desired relation.

This procedure is motivated by the fact that we want to have the same value on every branch of a fanout. Coverage is achieved with the same argument as before. The requirement of using unique variables for each run guarantees that assignments for the indexing variables can be chosen independently and merged without conflicts.

Similarly, we can handle bexprs with multiple outputs. We simply run the algorithm for each output independently. For this we introduce a unique indexing variable  $x_0^i$  for each output and require unique base names  $name^i$ . This ensures independence of the results and allows us to form the conjunction of the relations, where coverage follows as before.

## V. EFFICIENT PREIMAGE IMPLEMENTATION

The algorithm in Fig. 3 produces a partitioned abstraction relation, which allows early existential quantification to be

used in the preimage computations explained in Section II-A. We can, for example, use the technique described in [9]. But we can do even better by exploiting the special form that our relations have by virtue of how they are generated.

First, it is easy to show that the abstraction relation  $R[\mathcal{X}, \mathcal{T}]$  generated by our algorithm can be written as a conjunction  $S[\mathcal{X}] \wedge T[\mathcal{X}, \mathcal{T}]$ , where  $S[\mathcal{X}]$  contains no target variables. Terms in  $S[\mathcal{X}]$  are generated by line 3, when  $freevars(e) \subseteq \mathcal{C}$ , or by the consistency requirement in line 19 of Fig. 3. The conjunct  $T[\mathcal{X}, \mathcal{T}]$  has the form

$$\bigwedge_i (h_i[\mathcal{X}] \rightarrow t_i) \wedge (l_i[\mathcal{X}] \rightarrow \bar{t}_i),$$

where  $t_i$  ranges over the target variables  $\mathcal{T}$ , and no target variable occurs in any  $h_i$  or  $l_i$ . We define  $T_i[\mathcal{X}, t_i]$  to be  $(h_i[\mathcal{X}] \rightarrow t_i) \wedge (l_i[\mathcal{X}] \rightarrow \bar{t}_i)$ , so that  $T[\mathcal{X}, \mathcal{T}] = \bigwedge_i T_i[\mathcal{X}, t_i]$ .

The main theorem of this section shows how this structure is exploited. We first introduce some notation. Given a guard  $P$ , let  $\mathcal{F} = freevars(P)$  and let  $\mathcal{I}$  be the set of indices  $i$  such that  $t_i \in \mathcal{F}$ . Let  $R_{\downarrow P} = \bigwedge_{i \in \mathcal{I}} T_i$ . Intuitively,  $R_{\downarrow P}$  denotes the part of  $R$  that mentions the target variables in  $P$ . Note that  $P_{R_{\downarrow P}} = \exists \mathcal{F}. R_{\downarrow P} \wedge P$ .

*Theorem 2:* Given a guard  $P$  and a relation  $R$  produced by the algorithm in Fig. 3, define  $D = S \wedge \bigwedge_i \overline{h_i \wedge l_i}$ . Then:

$$P_R = \begin{cases} D \wedge P & \text{if } freevars(P) \cap \mathcal{T} = \emptyset \\ D \wedge \bar{l}_i & \text{if } P = t_i \\ D \wedge \bar{h}_i & \text{if } P = \bar{t}_i \\ D \wedge P_{R_{\downarrow P}} & \text{otherwise} \end{cases}$$

$$P^R = D \wedge \overline{P_R}$$

Because  $D$  does not depend on the guard  $P$ , it can be precomputed when  $R$  is created and used for all guards in the trajectory assertion to be transformed. By building hash tables that map  $t_i$  to  $h_i$  and  $l_i$ , the two middle cases can be computed very quickly. The fourth case, seen only rarely in practice, can be optimized using quantification scheduling.

The remainder of this section is devoted to proving the above theorem. First, given a relation  $R$ , we define  $dom(R)[\mathcal{X}] = \exists \mathcal{T}. R[\mathcal{X}, \mathcal{T}]$ . Intuitively,  $dom(R)$  holds for all valuations of the indexing variables that represent used, or consistent, indexing cases.

The following lemma states that  $dom(R)$  is equal to the expression  $D$  used in Theorem 2.

*Lemma 1:*  $dom(R) = S \wedge \bigwedge_i \overline{h_i \wedge l_i}$ .

*Proof:* By definition,  $dom(R) = \exists \mathcal{T}. R$ . Using the known shape of  $R$ , we expand this to  $\exists \mathcal{T}. S \wedge \bigwedge_i T_i$ . Since  $S$  does not depend on the target variables  $\mathcal{T}$ , and each  $T_i$  depends only on target variable  $t_i$ , we can push the quantifier inwards to obtain  $S \wedge \bigwedge_i \exists t_i. T_i$ . By eliminating the quantifier, this can easily be shown to equal  $S \wedge \bigwedge_i \overline{h_i \wedge l_i}$ .  $\square$

The next lemma states that only the parts of a relation that mention the target variables that occur in a guard need to be included when computing the preimage.

*Lemma 2:*  $P_R = dom(R) \wedge P_{R_{\downarrow P}}$ .

*Proof:* First, we express  $P_R$  in a form that resembles the goal. By definition,  $P_R = \exists T. R \wedge P$ . Using the known shape of  $R$ , this can be rewritten to  $\exists T. S \wedge (\bigwedge_i T_i) \wedge P$ . We then partition the variables  $T$  into the ones that occur in  $P$ ,  $\mathcal{U}$ , and the rest of them,  $\mathcal{V}$ , and also split the big conjunction similarly, to obtain  $\exists \mathcal{V}. \exists \mathcal{U}. S \wedge (\bigwedge_{i \notin \mathcal{I}} T_i) \wedge (\bigwedge_{i \in \mathcal{I}} T_i) \wedge P$ . Each  $T_i$  mentions only one target variable (namely  $t_i$ ), and  $S$  does not mention any target variables, so the quantifiers can be pushed inwards:  $S \wedge (\exists \mathcal{V}. \bigwedge_{i \notin \mathcal{I}} T_i) \wedge \exists \mathcal{U}. (\bigwedge_{i \in \mathcal{I}} T_i) \wedge P$ . So  $P_R = S \wedge (\exists \mathcal{V}. \bigwedge_{i \notin \mathcal{I}} T_i) \wedge P_{R \downarrow P}$  by the definition of  $R \downarrow P$ .

Now consider the subformula  $\exists \mathcal{V}. \bigwedge_{i \notin \mathcal{I}} T_i$ . Again, each  $T_i$  mentions only target variable  $t_i$ , so the quantifier can be pushed inwards to get  $\bigwedge_{i \notin \mathcal{I}} \exists t_i. T_i$ . It is easy to show that  $\exists t_i. T_i = \overline{h_i \wedge l_i}$ , and so  $\exists \mathcal{V}. \bigwedge_{i \notin \mathcal{I}} T_i = \bigwedge_{i \notin \mathcal{I}} \overline{h_i \wedge l_i}$ . We use this new equivalence to simplify the formula obtained earlier. Hence,  $P_R = S \wedge (\bigwedge_{i \notin \mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R \downarrow P}$ .

It remains to show  $S \wedge (\bigwedge_{i \notin \mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R \downarrow P} = \text{dom}(R) \wedge P_{R \downarrow P}$ . We do this by proving implication in both directions.

*Leftwards:* According to Lemma 1, the right side can be rewritten to  $S \wedge (\bigwedge_i \overline{h_i \wedge l_i}) \wedge P_{R \downarrow P}$ . This formula trivially implies the left side, since the only difference is that the left side contains fewer conjuncts.

*Rightwards:* Assume that  $S \wedge (\bigwedge_{i \notin \mathcal{I}} \overline{h_i \wedge l_i}) \wedge P_{R \downarrow P}$  holds. For a moment, we focus on the fact that the subformula  $P_{R \downarrow P}$  holds, which, by definition, is equal to  $\exists \mathcal{U}. (\bigwedge_{i \in \mathcal{I}} T_i) \wedge P$ . We can weaken this to the formula  $\exists \mathcal{U}. (\bigwedge_{i \in \mathcal{I}} T_i)$ , and push the quantifier inwards yielding  $\bigwedge_{i \in \mathcal{I}} \exists t_i. T_i$ . This formula can easily be shown to equal  $\bigwedge_{i \in \mathcal{I}} \overline{h_i \wedge l_i}$ .

Combining this formula with the original assumption results in  $S \wedge (\bigwedge_i \overline{h_i \wedge l_i}) \wedge P_{R \downarrow P}$ . By Lemma 1, this is equal to  $\text{dom}(R) \wedge P_{R \downarrow P}$ , which concludes the proof.  $\square$

Theorem 2 can now be established using Lemmas 1 and 2. It is easy to show that if a guard  $P$  contains no target variables, then  $P_R = P^R = \text{dom}(R) \wedge P$ . This justifies the first case of Theorem 2. For the second and third case of Theorem 2, we only need to specialize Lemma 2 with  $P = t_k$  and  $P = \overline{t_k}$ . The fourth case follows immediately from the lemmas. The identity  $P^R = \text{dom}(R) \wedge \overline{P_R}$  is easily shown using the definitions of the preimage operations.

One last observation allows us to improve the preimage calculations further. Notice that the expression  $D = \text{dom}(R) = \exists T. R[\mathcal{X}, T]$  is present as a conjunct in every case. As shown by Melham and Jones [7], the coverage condition  $\forall T. \exists \mathcal{X}. R[\mathcal{X}, T]$  must be supplied as an environmental constraint in the STE run. In other words, we are only interested in cases where this constraint is true. Since the formula  $D$  is implied by the constraint,  $D$  can safely be assumed to be true, and hence removed from the formulas. This significantly reduces the complexity of the preimage operations.

## VI. EXPERIMENTAL RESULTS

We illustrate the use of our algorithm on three classes of circuits; two that are traditional targets for the use of symbolic indexing and one that illustrates the power of our algorithm in applying abstraction in more subtle ways. All results were

obtained with BDD-based STE in the Forte [3] environment on a 2 Gbyte laptop.

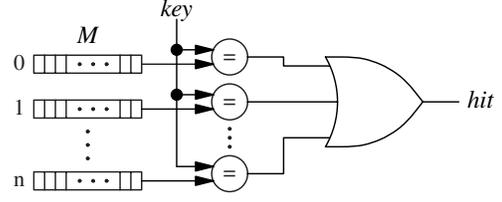


Fig. 4. High-level model of CAM.

Our first example is a traditional content-addressable memory (CAM), as illustrated in Fig. 4. The verification we are interested in is the behavior of the *hit* signal. It should be high iff there is an entry in the CAM with the same content as the key being presented. The specification function we use is the obvious one: try matching the key against each entry and take the disjunction of the results to yield the *hit* value. More precisely, the specification function we use is

$$\text{hit} = \bigvee_i (M[i] = \text{key})$$

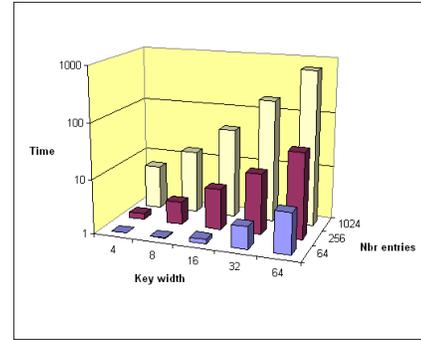


Fig. 5. Results for CAM verification.

We restricted the abstraction by declaring the key inputs to be symbolic constants. With this formulation, in Fig. 5 we show the time required for verifying a CAM of varying size and number of entries. The time for running the verification is split roughly in three equal parts: running the abstraction algorithm, computing the preimages, and finally running the STE verification. The results indicate that running time grows linearly with the size of the CAM. Indeed, our algorithm automatically generates the indexing suggested in [2], i.e., we yield an indexing that previously had to be developed with careful reasoning. It should be pointed out that this design could not be verified for larger sizes without using symbolic indexing: For BDD based verification, we ran out of memory; for SAT based verification, the verification process timed out.

It is interesting to note that if we apply our algorithm to the CAM specification but do not declare any symbolic constants, we obtain a much finer grained symbolic indexing scheme. For example, one case included in this finer indexing family

covers the case in which bit  $i$  of the key is different from bit  $i$  in every entry in the CAM. This input should lead to the hit signal being low. Not only does this approach yield a symbolic indexing scheme that is far less efficient, the STE verification failed on our design due to over-abstraction. Effectively, we obtained an X on the hit signal when we expected a 0. The reason for this was that in our design the key input to the CAM was protected by a (simulated) error correcting circuit. This circuit needed all inputs to the key to be fully defined to produce non-X outputs. In practice, it appears that such over-abstractions are relatively easy to avoid, but more work in this direction is clearly needed.

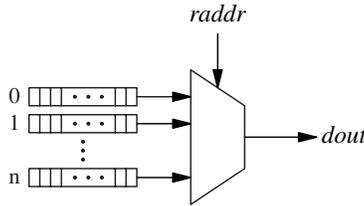


Fig. 6. High-level model of memory.

Our second example is a read operation for a memory, which is abstractly shown in Fig. 6. The specification function essentially consists of the natural sequence of “if read-address is  $i$  then return the content of address  $i$  else try  $i + 1$ ”. In contrast, the implementation uses a decoder circuit, pre-charged logic etc. to implement the read operation. Here, we state that the read address should not be abstracted, but kept fully symbolic. As can be seen in the graph in Fig. 7, the run-time behaviour of the STE verification is exceedingly good. A closer inspection of the symbolic indexing computed reveals that it is virtually identical to the one suggested in [2]. Given that we did not need to provide any information except stating that the address should be symbolic, this result clearly demonstrates the efficiency and practicality of our approach.

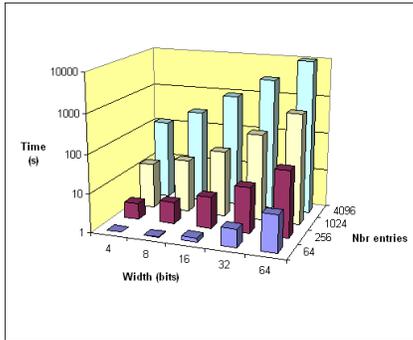


Fig. 7. Results for memory read.

Our final example is a scheduler abstractly shown in Fig. 8. Its functionality is to compute the address of the oldest entry that is ready. Providing a specification function that actually computes the oldest ready entry is fairly involved. However, supplying a relation that checks that a proposed address is the

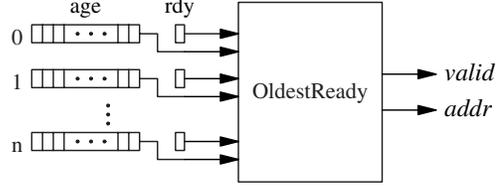


Fig. 8. High-level model of scheduler.

oldest ready is much easier. Thus we write the specification as a relation in terms of the symbolic constants: address  $\vec{a}$  and its age  $\vec{o}$ . The complete specification relation is given by:

$$\bigvee_i \left( \vec{a} = i \wedge rdy[i] \wedge age[i] = \vec{o} \wedge \bigwedge_{j \neq i} (\overline{rdy[j]} \vee age[j] < \vec{o}) \right)$$

In applying our algorithm, we state that  $\vec{a}$  and  $\vec{o}$  are symbolic constants. In Fig. 9 we show data on the run-time behavior of this verification effort, revealing that using this technique realistically sized schedulers can be verified. It is worth pointing out that trying to verify the same circuit without symbolic indexing, i.e., with variables in every state holding register and input, fails for circuits larger than 16 entries and width of the age registers of at least 4. In other words, not even a fairly trivial scheduler can be handled without abstraction. At the same time, with the method of this paper, this verification is completely straightforward and fully automatic.

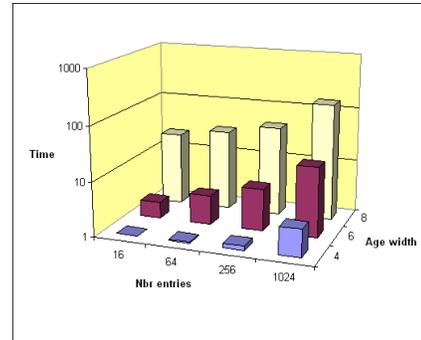


Fig. 9. Results for scheduler verification.

## VII. RELATED WORK

There is a rich and growing literature on automatic abstraction for model checking and verification. Some cornerstone techniques are *localization reduction* [10], *counterexample-guided abstraction refinement* [11], and *predicate abstraction* [12]. These and other methods have been extensively studied in the context of symbolic model-checking for hardware. By contrast, the literature on *automatic* abstraction for STE or GSTE is rather thin.

One notable exception is the work of Tzoref and Grumberg [13] on abstraction refinement for STE. This addresses the problem of over-abstraction in STE that manifests itself in symbolic simulation by a node carrying X where the

consequent expects 0 or 1. This is common with manually-formulated assertions, usually because some input variable node has not been driven by the antecedent [8]. A heuristic algorithm is presented in [13] that assigns fresh, distinct Boolean variables to input nodes in a clever way targeted at eliminating the Xs that make the model-checking run fail.

Roorda [14] presents a SAT-based method that assists in manual abstraction refinement for STE. The algorithm provides the user with hints for refining abstractions that over-approximate circuit behaviour. For a given assertion and circuit, Roorda's algorithm finds a minimal set of extra circuit inputs and an assignment of Booleans to them that will eliminate Xs at relevant circuit outputs. The user can then reformulate their assertion to drive these nodes with variables.

Both [13] and [14] are aimed at strengthening verification properties by driving more inputs. This refines the abstraction level of the specification to eliminate Xs, but in contrast with our work, does not in itself introduce complex, indexed families of abstractions.

Finally, our algorithm bears a resemblance to the D-algorithm [15] and other automatic test pattern generation methods. Both work backwards through a combinational circuit finding ways in which outputs might be forced high or low.

### VIII. CONCLUSION

The algorithm we presented in this paper is, to the best of our knowledge, the first automatic abstraction algorithm for STE that yields results equivalent to carefully hand-crafted assertions. This has the potential to significantly increase the use of this type of abstraction in STE-based verification.

Further improvement of our approach is planned. A fruitful field for future research will be to examine how to best encode the abstractions. That is, when should we reuse indexing variables, and when should we introduce fresh ones? This is particularly interesting when handling DAGs and multiple outputs, and when examining SAT-based STE verification. Some first results seem to indicate this type of abstraction does not speed up SAT-based STE verification the same way that it does BDD-based ones. Perhaps this is caused by an unsuitable encoding, or some entirely new approach is needed. We intend to investigate this issue much more thoroughly.

A second important research direction concentrates on how to refine the abstraction our algorithm computes in case it over-approximates the circuit function. In this paper, we introduced a simple and somewhat crude solution by allowing users to state that certain signals should not be abstracted. This is a perfectly workable solution for many examples, but more automatic abstraction refinement methods are needed to deploy our approach more widely. Applying the algorithm presented in [13], slightly modified so it can handle indexing, is one promising option to consider.

Finally, another obvious extension will be to incorporate the presented ideas into the GSTE algorithm [16]. We also intend to apply our method to a much wider class of problems to

determine its strengths and weaknesses. We suspect this will yield more insight into how to further enhance our approach.

### ACKNOWLEDGMENTS

This research was conducted while Carl-Johan Seger was a Visiting Fellow at Balliol College, Oxford University.

### REFERENCES

- [1] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, Mar. 1995.
- [2] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories using Symbolic Trajectory Evaluation," in *Design Automation Conference*. ACM Press, Jun. 1997, pp. 167–172.
- [3] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, Sept. 2005.
- [4] T. Schubert, "High Level Formal Verification of Next-Generation Microprocessors," in *DAC'03: Proceedings of the 40th conference on design automation*. ACM Press, 2003, pp. 1–6.
- [5] R. E. Bryant, "A Methodology for Hardware Verification Based on Logic Simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, Apr. 1991.
- [6] C.-T. Chou, "The Mathematical Foundation of Symbolic Trajectory Evaluation," in *Computer Aided Verification*, ser. LNCS, vol. 1633. Springer-Verlag, 1999, pp. 196–207.
- [7] T. F. Melham and R. B. Jones, "Abstraction by Symbolic Indexing Transformations," in *Formal Methods in Computer-Aided Design: FMCAD 2002*, ser. LNCS, vol. 2517. Springer-Verlag, 2002, pp. 1–18.
- [8] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical Formal Verification in Microprocessor Design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, Jul./Aug. 2001.
- [9] P. Chauhan, E. M. Clarke, S. Jha, J. H. Kukula, T. R. Shiple, H. Veith, and D. Wang, "Non-linear Quantification Scheduling in Image Computation," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press, 2001, pp. 293–298.
- [10] R. P. Kurshan, *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
- [11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
- [12] S. Graf and H. Saidi, "Construction of Abstract State Graphs with PVS," in *CAV'97*, ser. LNCS, vol. 1254. Springer-Verlag, 1997, pp. 72–83.
- [13] R. Tzoref and O. Grumberg, "Automatic refinement and vacuity detection for symbolic trajectory evaluation," in *Computer Aided Verification, 18th International Conference: CAV 2006*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 190–204.
- [14] J.-W. Roorda and K. Claessen, "SAT-based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation," in *Computer Aided Verification, 18th International Conference: CAV 2006*, ser. LNCS, vol. 4144. Springer-Verlag, 2006, pp. 175–189.
- [15] J. P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–291, Jul. 1966.
- [16] J. Yang and C.-J. H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 3, pp. 345–353, 2003.

# A Coverage Analysis for Safety Property Lists

Koen Claessen  
Chalmers University of Technology,  
Gothenburg, Sweden  
Email: koen@chalmers.se

**Abstract**—We present a coverage analysis that can be used in property-based verification. The analysis helps identifying "forgotten cases"; scenarios where the property list under analysis does not constrain a certain output at a certain point in time. These scenarios can then be manually investigated, possibly leading to new, previously forgotten properties being added. As there often exist cases in which outputs are not supposed to be specified, we also provide means for the specifier to annotate properties in order to control what cases are supposed to be unconstrained. Two main differences with earlier proposed similar analyses exist: The presented analysis is design-independent, and it makes an explicit distinction between intentionally and unintentionally underspecified behavior.

## I. INTRODUCTION

In property-based verification, a natural question that often arises is 'Have we specified enough properties?'

In simulation-based verification, there exist a myriad of *coverage* notions, that indicate how much of the design has been exercised during simulation. These notions can be used in order to decide when we have simulated "enough" (or rather when we have not yet simulated enough!)

When combining simulation-based verification with property-based verification, it is easy to see that such coverage notions do not help at all in deciding if we have actually specified enough properties. Two extreme cases help pointing this out: (1) It is possible to specify no properties at all, but still achieve a very high simulation coverage, (2) It is possible to achieve 100% simulation coverage (by for example exhaustive simulation or formal verification), but still properties might be missing.

There exist a number of notions of coverage in formal verification, where it is checked how much of the design under verification was actually needed in the formal proof [15], [5], [4], [6], [9]. One disadvantage of these methods is that they include the actual design in the coverage analysis.

In general, most coverage notions (for any analysis) are an over-approximation: When the stated coverage is high, it does not necessarily mean that the process at hand is finished, but when coverage is low, it usually means that concrete cases are missing. Therefore, when dealing with coverage in any situation, it is a good idea to provide several, complementary coverage notions. We would like to complement the arsenal of existing coverage notions for formal properties with an analysis that is design-independent. The reason for this is three-fold:

- In a design-dependent analysis, the complexity of the analysis becomes dependent on the complexity of the

design and the complexity of the properties. In a design-independent analysis, the complexity of the analysis is solely dependent on the properties, leading to a potentially much cheaper analysis.

- A design-dependent analysis can only be performed after the design has been completed. Moreover, the analysis will have to be re-done every time the design changes. In contrast, a design-independent analysis can start before the design is completed. Thus, a design-independent analysis fits in a design flow where properties and implementation evolve simultaneously in parallel. Properties can be analyzed and refined while the design is being worked on.
- Some property sets (for example specifications of industry standards) are supposed to be general, reusable, and thus design-independent. In this case it is vital that an analysis of these properties is design-independent. Many specifications are not strict, in the sense that they leave certain behavior underspecified (for example exactly how many clock cycles it takes until a ready signal is raised). General specifications should be as general as possible. The danger with using a design-dependent analysis is that the specification gets tied too much to the actual decisions that are made in the design at hand. At the same time, this is a challenge that a design-independent analysis needs to meet.

The idea of our analysis is simple: Given a list of safety properties, and a set of output signals from the design, our analysis checks if there exists a "forgotten case": a concrete trace where there exists a point in time where a particular output signal is not constrained by the properties. This trace represents a situation that the specifier might not have thought of when constructing the property list, and thus is a warning that the property list is not complete.

The presented coverage analysis works for all typical specification logics in which safety properties can be expressed. Here, we use a simple variant of LTL.

### A. An Example

As a small example, consider a circuit with an input *req* (for request) and an output *ack* (for acknowledge). A specification of the circuit is that whenever a request is made, an acknowledge should be made within two clock cycles. A property stating this in LTL might look as follows:

$$\Box(req \Rightarrow \text{next}(ack \vee \text{next} ack))$$

One might wonder is the above property is sufficient to specify the output *ack*. The answer is no; there are a number of "forgotten cases".

For example, the property does not say what should happen when no request has been made. What the specifier probably meant was that no acknowledge should be made unless a request has been made; a circuit that always outputs an acknowledge actually satisfies the above property. Our analysis warns about this situation by providing the following "forgotten case" – a concrete finite trace with an output that is unconstrained at a certain point in time:

$$\frac{req \mid 0 \ 0}{ack \mid 0 \ ?}$$

The question mark in the diagram indicates that no matter the value of *ack*, the property is satisfied. After studying such a forgotten case, the specifier can then refine the property list in order to outlaw certain behavior, making the specification more "tight". For example, a property like the following could be added:

$$\Box(\neg req \wedge \mathbf{next} \neg req \Rightarrow \mathbf{next} (\mathbf{next} \neg ack))$$

In other words, if no request has been made, then no acknowledge should be given.

Another example of a forgotten case is that the property list does not say that only one acknowledge should be given for each request. Our analysis produces the following forgotten case:

$$\frac{req \mid 1 \ 0 \ 0}{ack \mid 0 \ 1 \ ?}$$

This means that, at the place of the question mark, we are free to choose to acknowledge or not, without breaking any of the two properties. Again, the specifier can go back to the property list and decide if this should be allowed or not, by refining the property list.

The idea is that the specifier gets feedback about the property list, refines the list, gets more feedback, etc., until the analysis stops complaining, in which case there are no forgotten cases anymore.

### B. Forgotten cases

One interesting question is of course what the definition of a "forgotten case" is. One strong viewpoint could be that the property list should define the complete functional behavior of the circuit at hand. We feel that this restriction is too strong — many times the intention of a specification is to be general, not to have to decide the exact behavior.

For example, in the above example specification, it is unspecified when exactly the acknowledge should come, just that it should come within two clock cycles. This certainly does not completely specify the functional behavior of the circuit. As we shall see later, it turns out to be important to explicitly distinguish between *intentionally* and *unintentionally* underspecified behavior.

So, the choice of what a forgotten case is, is not so easy. It should be a balance between (1) providing useful information back to the user, and at the same time (2) not complaining too often.

The choice we have made in this paper is the following. A forgotten case is represented by two traces, both satisfying the property list at hand, but differing only at exactly one point in time for exactly one output signal. Then, since both traces satisfy the properties, this output is not constrained by the properties at this point in time, and indicates a forgotten case.

In other words, a forgotten case occurs in the following situation. Given an output  $x$  and a time point  $t$ , and given a trace where the values of all other signals and the values of all other points in time are known, and given that the list of properties holds, it is still not possible to decide what the value of  $x$  at time point  $t$  should be.

We discuss some alternative choices in section VII.

### C. Contributions

The contributions of this paper are the following.

- We introduce the concept of "forgotten case", that given a list of safety properties and an output signal, characterizes a point in time where the properties do not constrain this output.
- We present an analysis and implementation that finds forgotten cases, or shows their absence. The analysis is design-independent – the first of its kind as far as we know.
- To increase the usefulness of the analysis, we introduce the concept of "freeness" of an output signal, that allows a specifier to explicitly distinguish between intentionally and unintentionally underspecified signals. We argue that freeness is vital for design-independent analyses.

### D. Organization

In section II, we formally present what we mean by a forgotten case. In section III, we present a larger example, that illustrates the usefulness of the analysis. In sections IV and V, we use the example to illustrate why we need the concept of freeness, and present an adaptation of the semantics of forgotten case. In section VI, we show how we have implemented our analysis. Section VII concludes.

## II. SEMANTICS

In order to define more formally what a forgotten case is, we need to know a little bit more about the semantics of our properties. We use a simple variant of LTL in this paper, of which we can describe the syntax in the following way:

$$\phi, \psi ::= s \mid \neg\phi \mid \phi \wedge \psi \mid \mathbf{next} \phi \mid \Box\phi \mid \Diamond\phi \mid \Diamond!\phi$$

In the above,  $s$  is a signal name,  $\neg$  and  $\wedge$  are standard boolean operators (other operators can be implemented in terms of these),  $\mathbf{next}$  is the next operator, and  $\Box$  and  $\Diamond$  are the always and eventually operators. Finally, we also will use the

perhaps not so standard "eventually, exactly once" operator  $\diamond!$  in section VI.

To give semantics to the above syntax, we introduce the concept of *sequence*  $\sigma : \mathbb{N} \times \text{Signal} \rightarrow \mathbb{B}$ , which is a function from natural numbers and signal names to boolean values. The relation  $\models$  relates sequences and formulas in the following (standard) way:

$$\begin{array}{ll}
\sigma \models s & \text{iff. } \sigma(0, s) = 1 \\
\sigma \models \neg\phi & \text{iff. } \sigma \not\models \phi \\
\sigma \models \phi \wedge \psi & \text{iff. } \sigma \models \phi \text{ and } \sigma \models \psi \\
\sigma \models \text{next } \phi & \text{iff. } \sigma^1 \models \phi \\
\sigma \models \Box\phi & \text{iff. } \forall k. \sigma^k \models \phi \\
\sigma \models \Diamond\phi & \text{iff. } \exists k. \sigma^k \models \phi \\
\sigma \models \Diamond!\phi & \text{iff. } \exists!k. \sigma^k \models \phi
\end{array}$$

In the above, we write  $\sigma^k$  for the sequence  $\sigma$  shifted by  $k$  time steps;  $\sigma^k(t, s) = \sigma(t + k, s)$ . We use the quantifier  $\exists!$  which means "there exists a unique".

We should point out here that the main ideas in the paper can easily be adopted to other logics that are more complicated (for example Accellera's PSL [2]).

A *safety property* is a formula which only has finite counter examples. In other words, for a safety property  $\phi$ , if we have  $\sigma \not\models \phi$  for some  $\sigma$ , then there always exists a  $k$  such that for all sequences  $\sigma'$  which agree with  $\sigma$  on the first  $k$  points in time (i.e.  $\sigma'(i, s) = \sigma(i, s)$  for all  $0 \leq i \leq k$ ),  $\sigma'$  is also a counter example, i.e.  $\sigma' \not\models \phi$ .

#### A. Forgotten Cases

Given a property  $\phi$ , and an output signal  $s$ . A *forgotten case* for the output signal  $s$  is a pair of sequences  $(\sigma, \sigma')$  such that:

- $\sigma \models \phi$  and  $\sigma' \models \phi$
- for all signals  $s'$  which are not  $s$ ,  $\forall t. \sigma(t, s') = \sigma'(t, s')$
- $\exists!t. \sigma(t, s) \neq \sigma'(t, s)$

In other words, both  $\sigma$  and  $\sigma'$  satisfy the property  $\phi$ , but differ at only exactly one point in time, and only for the signal  $s$ .

Because of the large overlap between the sequences  $\sigma$  and  $\sigma'$  we usually depict a forgotten case as one sequence, with a question mark at the place of the difference point.

A forgotten case for a list of properties  $\phi_i$  is simply a forgotten case for the conjunction of all  $\phi_i$ .

The next section tries to motivate forgotten cases by means of a larger example.

### III. A LARGER EXAMPLE: A FIFO

Consider the following specification of a simple FIFO, depicted in Fig. 1. The input signals are *get*, *put* and a vector *in*, and the output signals are *err* and vectors *fst* and *num*. For simplicity, we specify that putting takes priority over getting. When we try to put something in a full FIFO, or get something from an empty FIFO, the signal *err* becomes 1 for one clock cycle. The output *fst* always indicates the first element of the FIFO, and the output *num* indicates the number of elements currently in the FIFO (maximum  $n$ ).

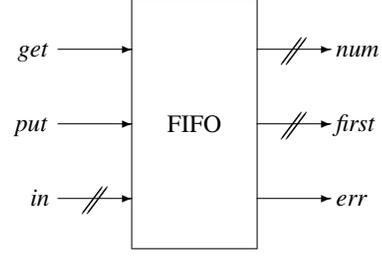


Fig. 1. A simple FIFO interface

An initial attempt to create a list of safety properties formalizing the above description might look as follows.

$$\begin{array}{ll}
\Box(\text{put} = 1 \wedge \text{num} = n & \Rightarrow \text{next } \text{err} = 1) \\
\Box(\text{put} = 1 \wedge \text{num} < n & \Rightarrow \text{next } \text{num} = \text{num} + 1 \\
& \wedge \text{next } \text{err} = 0) \\
\Box(\text{put} = 1 \wedge \text{num} = 0 & \Rightarrow \text{next } \text{fst} = \text{in}) \\
\Box(\text{put} = 1 \wedge 0 < \text{num} < n & \Rightarrow \text{next } \text{fst} = \text{fst}) \\
\Box(\text{get} = 1 \wedge \text{put} = 0 \wedge \text{num} = 0 & \Rightarrow \text{next } \text{err} = 1) \\
\Box(\text{get} = 1 \wedge \text{put} = 0 \wedge 0 < \text{num} & \Rightarrow \text{next } \text{num} = \text{num} - 1 \\
& \wedge \text{next } \text{err} = 0)
\end{array}$$

We can now analyze this list of properties using the proposed analysis, in order to discover forgotten cases in our specification. Note that we are only analyzing the list of properties, not the design. At this stage, having the design ready for formal verification is not necessary.

When we ask the analysis about the property coverage of output *err*, it immediately replies that *err* is not constrained by the properties at time point 1, as the following trace shows:

<i>get</i>	0
<i>put</i>	0
<i>in</i>	0
<i>num</i>	0
<i>fst</i>	0
<i>err</i>	?

No matter what the value of the ? in the trace, the property list is still fulfilled. Indeed we should have added a property  $\text{err} = 0$  at time 0!

After this, the analysis still complains; *err* is unconstrained when we do not put or get something from the FIFO:

<i>get</i>	0	0
<i>put</i>	0	0
<i>in</i>	0	0
<i>num</i>	0	0
<i>fst</i>	0	0
<i>err</i>	0	?

And indeed, we should have added a property that says that errors do not occur when we do not try to change the contents of the FIFO:

$$\Box(\text{get} = 0 \wedge \text{put} = 0 \Rightarrow \text{next } \text{err} = 0).$$

Now, the analysis stops complaining about *err*.

Next, we analyze the output *num*. We find out that *num* is not constrained in the first point in time either:

<i>get</i>	0
<i>put</i>	0
<i>in</i>	0
<i>num</i>	?
<i>fst</i>	0
<i>err</i>	0

This counter example leads to us adding the property  $num = 0$  at time 0. Next, the analysis complains about *num* being unconstrained when we do not put or get:

<i>get</i>	0	0
<i>put</i>	0	0
<i>in</i>	0	0
<i>num</i>	0	?
<i>fst</i>	0	0
<i>err</i>	0	0

This is easily fixed by adding the property:

$$\square((get = 0 \wedge put = 0) \Rightarrow \text{next } num = num).$$

However, the analysis still thinks there is a forgotten case. It reports:

<i>get</i>	1	0
<i>put</i>	1	0
<i>in</i>	0	0
<i>num</i>	0	?
<i>fst</i>	0	0
<i>err</i>	0	1

In other words, when an error occurs, it is not specified what should happen with *num*. We fix this by adapting the last property we added thus:

$$\square((get = 0 \wedge put = 0) \vee \text{next } err = 1 \Rightarrow \text{next } num = num).$$

Finally, the analysis stops complaining about the output *num*.

#### IV. FREE SIGNALS

Dealing with the third output signal, *fst*, there appear to be two problems. Firstly, it is not always the case that we want to specify what the value of a signal is in all cases. For example, when the FIFO is empty, we would like to leave *fst* unspecified, since there is no first value in the FIFO. At the moment, the analysis would simply always complain about this, making it rather useless.

Secondly, sometimes it is hard or impossible to completely formally specify the exact behavior of a particular signal in a temporal logic, and as a specifier one wants to be able to take the pragmatic decision of not specifying the behaviour completely. Again, the analysis would immediately find holes in the specification, holes which have deliberately been put there. In the case of the signal *fst*, formally specifying the

exact FIFO behavior for general *n* is impossible in a limited logic like LTL.

One solution to this problem is simply not to use the analysis on the output *fst*. This has an obvious drawback, namely that we will not be able to find *real* forgotten cases, as opposed to the intended forgotten cases that we already know about. Instead, we would like to have a more fine-grained choice than the choice between either fully specifying a signal, or not checking coverage for that signal at all.

So, we would like to argue for another solution, namely one where the specifier explicitly indicates in what cases an output is allowed to be unconstrained. We therefore introduce a new construct **free** *s* to the specification logic, that can be used to express that the output *s* is allowed to be unconstrained. As a logical construct, **free** *s* is simply true, but to the analysis, it is a way to suppress complaints about the signal *s*.

For example, our analysis complains about the output *fst* being unconstrained in the beginning, when the FIFO is empty. This can be remedied by adding the following property:

$$\square(num = 0 \Rightarrow \text{free } fst).$$

The above property explicitly expresses the unconstrainedness of *fst* in the case when  $num = 0$ . Its effect is that this case is not complained about anymore.

A misunderstanding that easily arises when being presented with **free** for the first time is to think that adding a property specifying that **free** *s* should be true means that the properties should hold for any value of *s*. This is not what free means, and would actually make the logic non-monotonic! Instead, **free** *s* should be interpreted as "the signal *s* is allowed to take on more than one value here", and it does not logically constrain the value of *s* at all. It is simply a way of saying to the analysis, "yes, I know *s* can have multiple values in this case, but I decide that this is OK". In the next section, the semantics of **free**, and its relation to forgotten cases, is specified more formally.

After adding the above property, our analysis complains about the output *fst* being unconstrained when we put two elements in the FIFO, and get an element out once<sup>1</sup>:

<i>get</i>	0	0	1	0
<i>put</i>	1	1	0	0
<i>in</i>	17	5	0	0
<i>num</i>	0	1	2	1
<i>fst</i>	0	17	17	?
<i>err</i>	0	0	0	0

And indeed, we have not said anything about this particular case. This actually is the simplest situation where it matters if the circuit implements a FIFO or something else, like a LIFO. If the specifier decides not to specify the exact FIFO behavior, this can be expressed by adding **next free** *fst* to the right-hand side of the last property:

<sup>1</sup>The actual counter example that was generated was slightly edited for presentational reasons.

$$\begin{aligned} \Box(\text{get} = 1 \wedge \text{put} = 0 \wedge 0 < \text{num} &\Rightarrow \text{next } \text{num} = \text{num} - 1 \\ &\wedge \text{next } \text{err} = 0 \\ &\wedge \text{next } \text{free } \text{fst} \end{aligned}$$

In other words, when we remove an element from the FIFO, we are not quite sure what the new first element is going to be.

Even now, our analysis still reports a forgotten case, namely when we do not put or get at all, *fst* is unconstrained:

<i>get</i>	0	0
<i>put</i>	0	0
<i>in</i>	0	0
<i>num</i>	0	0
<i>fst</i>	0	?
<i>err</i>	0	0

We solve this by adding  $\text{next } \text{fst} = \text{fst}$  to the right-hand side of the second property we added when we analyzed *num*:

$$\Box((\text{get} = 0 \wedge \text{put} = 0) \vee \text{next } \text{err} = 1 \Rightarrow \text{next } \text{num} = \text{num} \wedge \text{next } \text{fst} = \text{fst}).$$

After adding this property, the analysis stops complaining.

Note that, without **free**, our only option would have been to not analyze the output *fst* at all, which would have meant not finding the last forgotten case, which turned out to be a "real" forgotten case.

The final, refined, property list now looks as follows.

$$\begin{aligned} \text{err} = 0 \wedge \text{num} = 0 & \\ \Box(\text{put} = 1 \wedge \text{num} = n &\Rightarrow \text{next } \text{err} = 1) \\ \Box(\text{put} = 1 \wedge \text{num} < n &\Rightarrow \text{next } \text{num} = \text{num} + 1 \\ &\wedge \text{next } \text{err} = 0) \\ \Box(\text{put} = 1 \wedge \text{num} = 0 &\Rightarrow \text{next } \text{fst} = \text{in}) \\ \Box(\text{put} = 1 \wedge 0 < \text{num} < n &\Rightarrow \text{next } \text{fst} = \text{fst}) \\ \Box(\text{get} = 1 \wedge \text{put} = 0 \wedge \text{num} = 0 &\Rightarrow \text{next } \text{err} = 1) \\ \Box(\text{get} = 1 \wedge \text{put} = 0 \wedge 0 < \text{num} &\Rightarrow \text{next } \text{num} = \text{num} - 1 \\ &\wedge \text{next } \text{err} = 0 \\ &\wedge \text{next } \text{free } \text{fst}) \\ \Box(\text{get} = 0 \wedge \text{put} = 0 &\Rightarrow \text{next } \text{err} = 0) \\ \Box((\text{get} = 0 \wedge \text{put} = 0) \vee \text{next } \text{err} = 1 &\Rightarrow \text{next } \text{num} = \text{num} \\ &\wedge \text{next } \text{fst} = \text{fst}) \\ \Box(\text{num} = 0 &\Rightarrow \text{free } \text{fst}) \end{aligned}$$

The reader can compare this list with the list given at the beginning of section III.

## V. SEMANTICS OF FREE SIGNALS

With the introduction of the new construct **free**, we need to adapt our semantics accordingly. First, we add **free** to the syntax:

$$\phi, \psi ::= \dots \mid \text{free } s$$

Note that **free** can only be used on signal names, not on formulas in general.

To give semantics to **free**, we first slightly adapt the concept of sequence; a sequence  $\sigma$  now also determines the value of every **free** *s*. Let **FreeSignal** be the set  $\{\text{free } s \mid s \in \text{Signal}\}$ . Then,  $\sigma : \mathbb{N} \times (\text{Signal} \cup \text{FreeSignal}) \rightarrow \mathbb{B}$ . We adapt the relation  $\models$  in the following way:

$$\sigma \models \text{free } s \text{ iff. } \sigma(0, \text{free } s) = 1$$

We adapt the notion of forgotten case in the following way. Given a property  $\phi$ , and an output signal *s*, a *forgotten case* for the output signal *s* is a pair of sequences  $(\sigma, \sigma')$  such that:

- $\sigma \models \phi$  and  $\sigma' \not\models \phi$
- for all signals  $s'$  which are not *s*,  $\forall t. \sigma(t, s') = \sigma'(t, s')$
- $\exists! t. \sigma(t, s) \neq \sigma'(t, s)$  and  $\sigma(t, \text{free } s) = 0$

In other words,  $\sigma$  and  $\sigma'$  differ at exactly one point in time, and only for the signal *s*, and **free** *s* has to be 0 at that point in time. Thus, if **free** *s* has the value 1 at a point in time, then there can not be a forgotten case that makes *s* different at that point in time.

A final remark we can make here is that the addition of the **free** construct is possible in any logic that resembles LTL. One alternative way to deal with *free* is to introduce a special new signal *free\_s* for every signal *s*. Then, the syntax does not have to change, and the semantics of the logic does not change either. The only part that changes is the definition of forgotten case.

## VI. IMPLEMENTATION

Given a property list and an output signal *s*, our analysis either finds a forgotten case for *s*, or shows that no forgotten case exists. The implementation of the analysis is quite straightforward. The question that the analysis answers is: Given a property list, and a specific output *s*, do there exist two sequences  $\sigma_1$  and  $\sigma_2$  that both satisfy the properties, and that have exactly the same values for all signals at all points in time, except for exactly 1 point in time, where the value of *x* differs?

### A. General implementation

Let us assume that we have access to a model checker for the logic at hand. Any model checker will do – in our implementation, we used SMV [11].

Let us first present the analysis without the involvement of the **free** operator. Suppose the property list is the formula  $\phi$ . We make two copies of  $\phi$ , namely  $\phi(s)$  and  $\phi(s')$ ; in the second copy we have replaced all occurrences of *s* by a fresh signal  $s'$ . Then, we ask the model checker to find a sequence that satisfies the following formula:

$$\phi(s) \wedge \phi(s') \wedge \Diamond!(s \neq s')$$

The operator  $\Diamond!$  can be implemented by either adding a little bit of circuitry, or by expressing it in terms of the standard LTL until operator, for example by setting  $\Diamond!a = \neg a \mathbf{U} (a \wedge \text{next } \Box \neg a)$ .

It is easy to see that a sequence satisfying the above formula corresponds exactly to a forgotten case. If no such sequence exist, then there are no forgotten cases.

### B. Using free signals

When free signals are involved, we again make two copies of  $\phi$ , namely  $\phi(s, \text{free } s)$  and  $\phi(s', \text{free } s)$ . Again, we replace  $s$  by  $s'$  (but we do not have to replace  $\text{free } s$  by  $\text{free } s'$  because  $\text{free } s'$  does not affect the definition of forgotten case).

We then ask the model checker to find a sequence satisfying the following formula. We have replaced  $\text{free } s$  by a fresh signal  $\text{free}_s$ , because the model checker does not know of the construct  $\text{free}$ .

$$\Box(\phi(s, \text{free}_s) \wedge \phi(s', \text{free}_s) \wedge \Diamond!(\neg \text{free}_s \wedge s \neq s'))$$

Again, a sequence satisfying the above formula corresponds exactly to a forgotten case, and if no such sequence exist, then there are no forgotten cases.

### C. Implementation using observers

We have also implemented the analysis in our Lava framework [7]. In Lava, safety properties are specified using a *safety property observer*. A safety property observer is sometimes also called "monitor circuit" or "checker circuit"; a circuit that has as inputs all signals appearing in the properties, and that has only one output "OK", that is always high if and only if the properties hold. Observers can be constructed automatically for formulas in many logics [13], [14], [1]. The concept of safety property observer makes it possible to use different specification logics in the same framework.

The following implementation technique can be used for any specification logic whose formulas can be translated into safety property observers. We start by building the safety property observer belonging to the property list [12].

Then, we build a circuit using two copies of the observer, one observer that monitors all signals and the output  $s$ , and one observer that monitors the same signals, but instead of  $s$  monitors a new fresh signal  $s'$ . We denote the output of the first observer by  $\text{OK}(s, \text{free}_s)$ , and the output of the second observer by  $\text{OK}(s', \text{free}_s)$ .

We then ask the model checker to find a trace that satisfies:

$$\Box(\text{OK}(s, \text{free}_s)) \wedge \Box(\text{OK}(s', \text{free}_s)) \wedge \Diamond!(\neg \text{free}_s \wedge s \neq s')$$

The size of the analyzed circuit is linear in the size of the property observer. The model checking problems that are produced by the above process very seldom pose a challenge to standard LTL model checkers.

## VII. DISCUSSION AND CONCLUSION

There are basically three reasons for not fully specifying the behaviour of all output signals: (1) The output is supposed to be underconstrained in the specification; (2) By choice, the specifier has decided to leave the output underconstrained; (3) The specifier has forgotten a case. We argue for an analysis that can discover the 3rd case, by forcing the specifier to explicitly document in the property list if cases (1) or (2) are meant. We believe that it is a Good Thing to explicitly distinguish between cases that have been forgotten by the specifier and cases that are supposed to be underconstrained. We believe that this leads to specifications of higher quality,

which in turns leads to more dependable verification results. Lastly, the analysis can be used in both simulation-based and formal property verification.

We would like to contrast property-based verification against reference-model-based verification. In reference-model-based verification (applicable also to equivalence checking), the specification of a circuit consists of a reference model, which often is implemented in a higher-level formalism than the actual circuit. In this case, the model provides in principle a complete specification of the circuit, since the model specifies for each situation (perhaps modulo details such as time-based refinements and such) what the behavior of the circuit should be.

In property-based verification, the advantage is a more incremental, compositional way of specifying what a circuit is supposed to do. One can start with some simple properties (perhaps reused from property libraries), and adding more and more desired behavior by means of more properties. While this has many advantages, the drawback is that it is hard to know when to stop. The contribution of this paper is to alleviate this problem by providing an automatic means of suggesting what properties could possibly be missing, and by forcing the specifier to decide whether certain missing behaviors intentionally should be left underspecified or not. The analysis is complementary to other property analyses such as vacuity checking [10], [3].

### A. Results

We have applied the presented analysis to a variety of different types of designs: Two different types of arbiters, three different types of FIFOs (sizes: 32 16-bit entries), one ALU (16-bit integers), a floating-point adder (32-bit), a memory controller, and a CAM (128 entries). In all cases, we started with a property list designed by ourselves, and in all cases we found at least one forgotten case, which lead to the addition or adaptation of a property. All property lists were in the end deemed OK by the analysis (i.e. the analysis did not find any forgotten cases anymore). (To be honest, it is hard to quantify what this means since we also added  $\text{free}$  statements about some signals.)

The analysis usually completes in a few seconds, sometimes up to a minute. In one case, the case for the ALU, we had to "cheat". The specification contained a reference to multiplication. Our model checker could unfortunately not really deal with the multiplier that was contained in the property observer. Instead, we replaced multiplication by another, easier function, for the sake of the analysis. This was not really satisfactory, but it was obvious to us that this would not change the result of the analysis.

### B. Related work

Much work has been done on coverage analyses that are performed post-formal verification, such as [15], [5], [4], [6], [9]. These analyses involve the design as well, and often analyze what happened during the actual formal verification process. Our analyses complements these by (1) providing an

analysis that is design-independent, and (2) provides concrete traces indicating forgotten cases rather than (for example) points in the design that are not needed in the formal proof. Cockler et al. [5] mention the idea of *don't cares* in measuring coverage, something akin to our *free*.

The work that comes closest to what we present here is the work by Große et al. [9]. They also find a concrete trace where a signal differs at a particular point in time. Two main differences exist between their work and ours: (1) Their analysis is design-dependent, and (2) their analysis only works for a verification environment where bounded model checking is used. Their concept of coverage is thus defined up to a certain bound.

Das et al. [8] present a design-independent analysis for property-based verification with a similar goal in mind as ours. Using a stuck-at 0/1 fault model for inputs and outputs, they calculate a measure that expresses how well such faults are covered by a given property list.

### C. Future work

Our definition of what a forgotten case is, is currently a very simple one, but also quite restricted. We plan to investigate variants of the choice we made. Two relaxations come to mind: (A) the two sequences of a forgotten cases can differ at multiple points in time for the analyzed signal  $s$ , (B) the two sequences of a forgotten cases can differ for multiple output signals at multiple points in time. It is easy to see that definition (B) is the "most general" one. It is also clear that definition (A) will lead to more complaints than the analysis presented in the paper, and definition (B) to even more complaints. Our initial experiments here show that loosening the definition of forgotten case easily leads to a situation where the analysis complains too often – there are many cases where the analysis complains, but where the specifier neither feels that the analysis has pointed out a useful case, nor knows how to refine the property list to make the complaint disappear. This is an issue that needs further investigation.

Other future work is to investigate the similarities between Große et al. [9] and our work. We have already started to develop a framework where their ideas (comparing variants of the circuit with the specification) and our ideas (comparing a variant of the specification with the specification itself) can be both expressed.

Finally, we need to investigate if the analysis can be extended to non-safety properties. This seems a trivial task at first, but some details need to be resolved before this can be done.

### REFERENCES

- [1] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification (CAV)*, 2000.
- [2] Accellera. Accellera Property Specification Language, Reference Manual, 2005. <http://pslsugar.org>.
- [3] Doron Bustan, Alon Flaisher, Orna Grumberg, Orna Kupferman, and Moshe Y. Vardi. Regular vacuity. In *Correct Hardware Design and Verification Methods (CHARME)*, 2005.
- [4] Hana Chockler, Orna Kupferman, Robert P. Kurshan, and Moshe Vardi. A practical approach to coverage in model checking. In *Computer Aided Verification (CAV)*, 2001.
- [5] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics for temporal logic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2001.
- [6] Hana Chockler, Orna Kupferman, and Moshe Y. Vardi. Coverage metrics in formal verification. In *Correct Hardware Design and Verification Methods (CHARME)*, 2003.
- [7] K. Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Chalmers University of Technology, 2001.
- [8] Sayantan Das, Ansuman Banerjee, Prasenjit Basu, Pallab Dasgupta, P. Chakrabarti, Chunduri Rama Mohan, and Limor Fix. Formal methods for analyzing the completeness of an assertion suite against a high-level fault model. In *The 18th International Conference on VLSI Design (VLSID)*, pages 201–206. IEEE Computer Society, 2005.
- [9] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *Design, Automation and Test in Europe (DATE)*, 2007.
- [10] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, 1999.
- [11] K. McMillan. The SMV model checker, 2002. <http://www.kenmcml.com/smv.html>.
- [12] F. Lagnier N. Halbwachs and P. Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*, 1993.
- [13] J.-C. Fernandez N. Halbwachs and A. Bouajjani. An executable temporal logic to express safety properties and its connection with the language lustre. In *Sixth International Symp. on Lucid and Intensional Programming (ISLIP'93)*, 1993.
- [14] P. Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming (ICALP'96)*, LNCS 1099. Springer Verlag, 1996.
- [15] O. Grumberg S. Katz. Have i written enough properties? – a method of comparison between specification and implementation. In *Correct Hardware Design and Verification Methods (CHARME)*, 1999.

# What Triggers a Behavior?

Orna Kupferman and Yoav Lustig  
 School of Engineering and Computer Science  
 Hebrew University, Jerusalem, 91904, Israel  
 Email: {orna,yoadl}@cs.huji.ac.il

**Abstract**—We introduce and study *trigger querying*. Given a model  $M$  and a temporal behavior  $\varphi$ , trigger querying is the problem of finding the set of scenarios that trigger  $\varphi$  in  $M$ . That is, if a computation of  $M$  has a prefix that follows the scenario, then its suffix satisfies  $\varphi$ . Trigger querying enables one to find, for example, given a program with a function  $f$ , the scenarios that lead to calling  $f$  with some parameter value, or to find, given a hardware design with signal  $err$ , the scenarios after which the signal  $err$  ought to be eventually raised.

We formalize trigger querying using the temporal operator  $\mapsto$  (triggers), which is the most useful operator in modern industrial specification languages. A regular expression  $r$  triggers an LTL formula  $\varphi$  in a system  $M$ , denoted  $M \models r \mapsto \varphi$ , if for every computation  $\pi$  of  $M$  and index  $i \geq 0$ , if the prefix of  $\pi$  up to position  $i$  is a word in the language of  $r$ , then the suffix of  $\pi$  from position  $i$  satisfies  $\varphi$ . The solution to the trigger query  $M \models? \mapsto \varphi$  is the maximal regular expression that triggers  $\varphi$  in  $M$ . Trigger querying is useful for studying systems, and it significantly extends the practicality of traditional query checking [6]. Indeed, in traditional query checking, solutions are restricted to propositional assertions about states of the systems, whereas in our setting the solutions are temporal scenarios.

We show that the solution to a trigger query  $M \models? \mapsto \varphi$  is regular, and can be computed in polynomial space. Unfortunately, the polynomial-space complexity is in the size of  $M$ . Consequently, we also study *partial trigger querying*, which returns a (non empty) subset of the solution, and is more feasible. Other extensions we study are *observable trigger querying*, where the partial solution has to refer only to a subset of the atomic propositions, *constrained trigger querying*, where in addition to  $M$  and  $\varphi$ , the user provides a regular constraint  $c$  and the solution is the set of scenarios respecting  $c$  that trigger  $\varphi$  in  $M$ , and *relevant trigger querying*, which excludes vacuous triggers — scenarios that are not induced by a prefix of a computation of  $M$ . Trigger querying can be viewed as the problem of finding sufficient conditions for a behavior  $\varphi$  in  $M$ . We also consider the dual problem, of finding necessary conditions to  $\varphi$ , and show that it can be solved in space complexity that is only logarithmic in  $M$ .

## I. INTRODUCTION

The field of formal verification developed from the need to verify that a system satisfies its specification. Since its conception, the field has enjoyed great progress in the development of practical tools and better understanding of the problems and models related to formal verification. One of the concepts that has emerged in the context of formal verification is that of *model exploration*. The idea, as first noted by Chan in [6], is that, in practice, model checking is often used for understanding the system rather than for verifying its correctness.

Chan suggested to formalize model exploration by means of *query checking*. The input to the query-checking problem is a model  $M$  and a query  $\varphi$ , where a query is a temporal-logic formula in which some proposition is replaced by the place-holder “?” (e.g.,  $AG?$ ). A solution to the query is a propositional assertion that, when it replaces the place-holder, results in a formula that is satisfied in  $M$ . For example, if the query is  $AG?$ , then the set of solutions include all assertions  $\psi$  for which  $M \models AG\psi$ . A query checker should return the strongest solutions to the query (strongest in the sense that they are not implied by other solutions).<sup>1</sup> The work of Chan was followed by further work on query checking, studying its complexity, cases in which only a single strongest solution exists, the case of multiple (possibly related) place-holders, and more [4], [8], [15], [7].

We believe that model exploration, and in particular query checking, is a very natural and interesting task. Query checking suffers, however, from a serious shortcoming: The result of a query check is a propositional assertion. Thus, query checking is restricted to questions regarding one point in time, whereas most interesting questions about systems involve scenarios that develop over time.

Consider, for example, a programmer trying to understand the code of some computer program. In particular, the programmer is interested in situations in which some function is called with some parameter value. The actual state in which the function is called is by far less interesting than the scenario that has led to it. Query checking does not enable us to reveal such scenarios.

In this work we introduce and study *trigger querying*, which addresses the shortcoming described above. Given a model  $M$  and a temporal behavior  $\varphi$ , trigger querying is the problem of finding the set of scenarios that trigger  $\varphi$  in  $M$ . That is, the set of scenarios such that if a computation of  $M$  has a prefix that follows a scenario in the set, then its suffix satisfies  $\varphi$ .

We formalize trigger querying using the temporal operator  $\mapsto$  (triggers). The trigger operator was introduced in SUGAR (the precursor of PSL [3], called suffix implication there). We use the name trigger suggested in ForSpec [1] as it is more indicative of the operator meaning. System Verilog Assertions (SVA) [17] is another popular industrial specification formalism in which the operator triggers plays an important role. Consider a system  $M$  with a set  $AP$  of atomic propositions. A word  $w$  over the alphabet  $2^{AP}$  triggers an LTL formula  $\varphi$  in

<sup>1</sup>Note that a query may not only have several solutions, but may also have several strongest solutions.

the system  $M$ , denoted  $M \models w \mapsto \varphi$ , if for every computation  $\pi$  of  $M$ , if  $w$  is a prefix of  $\pi$ , then the suffix of  $\pi$  from position  $|w|$  satisfies  $\varphi$  (note that there is an “overlap” and the  $|w|$ -th letter of  $\pi$  participates both in the prefix  $w$  and in the suffix satisfying  $\varphi$ .) The solution to the trigger query  $M \models? \mapsto \varphi$  is the set of words  $w$  that trigger  $\varphi$  in  $M$ . Since, as we show, the solution is regular, trigger-querying algorithms return the solution by means of a regular expression or an automaton on finite words.

Let us consider an example. Assume that  $M$  models a hardware design with a signal  $err$  that is raised whenever an error occurs. We might be interested in characterizing the scenarios after which the signal  $err$  is raised. This is, exactly the set of scenarios that trigger  $err$  — the solution to the trigger query  $M \models? \mapsto err$ . It may also be the case that we are really interested in characterizing the scenarios after which  $err$  ought to be raised. The difference is that now we are interested in “crossing the point of no return”; that is, the point from which  $err$  would eventually (possibly in the distant future) be raised. The set of such scenarios are the solution to the trigger query  $M \models? \mapsto Ferr$ .

Another way to see the importance of the extension of query checking from a propositional to a temporal setting is to go back to the context of model checking. It is widely acknowledged that if a bug is found, it should be reported with a temporal counter example. Indeed, counter examples allow the user to see the bug in context and to understand what has caused the bug and how to fix it. This corresponds to the model explorer need to see full scenarios rather than states. Getting from a query checker the propositional assertions that are the solutions to the query  $? \rightarrow Ferr$  (or even to  $G(? \rightarrow Ferr)$ ) is much less informative than getting the full scenarios that lead to  $err$ .<sup>2</sup>

We solve trigger querying and show that the problem is tight for polynomial space. Unlike LTL model-checking, whose complexity is also polynomial space, here the polynomial-space complexity is not only in the length of the specification but also in the size of the system. Consequently, we consider a more feasible version of trigger querying. The idea is that when the user cannot get a complete characterization of the scenarios triggering a behavior, he may still be interested in getting examples of words triggering the behavior. In *partial trigger querying*, the algorithm returns a subset of the solution to the trigger query (unless the complete solution is empty, the subset should not be empty). The complexity demands of partial trigger querying are indeed lower than these of trigger querying. Specifically, the complexity in the system is nondeterministic polynomial time rather than polynomial space. Beyond the lower complexity, partial trigger querying can be implemented symbolically, and we describe BDD-based and SAT-based algorithms for solving trigger querying.

In addition to trigger querying as presented above, we

<sup>2</sup>Note that temporal querying cannot be reduced to a search for counter examples. For example, the solution to  $M \models? \mapsto Xerr$  is the set of words  $w$  such that all the computations of  $M$  that start with  $w$  would reach  $err$  in their next cycle. On the other hand, the counterexamples to  $M \models G \neg Xerr$  are words  $w$  such that there is a computation of  $M$  that starts with  $w$  and reaches  $err$  in its next cycle; such words  $w$  do not necessarily trigger  $Xerr$ .

introduce and study several natural variants of the problem. First, suppose that a finite word  $w$  cannot be generated by the system  $M$  (i.e., it is not a prefix of a computation of  $M$ ). Then,  $w$  satisfies the query  $M \models? \mapsto \varphi$  in a vacuous way, as indeed, every computation of  $M$  that has  $w$  as a prefix continues to a suffix satisfying  $\varphi$ . A user, however, is rarely interested in seeing such vacuous triggers. In *relevant trigger querying*, we exclude vacuous triggers, and the solution to a relevant trigger query is restricted to words generated by the system.

The next variant we consider is *constrained trigger querying*. Model exploration is usually not a specific question to which there is a definite answer but rather an open-ended activity. Accordingly, trigger querying does not consist of a single query but rather it is an interactive dialog between the user and the trigger-query tool. A natural course of events is one in which the user refines the trigger queries in order to find scenarios that not only trigger the behavior in question, but also satisfy some constraints. For example, the user may search for scenarios that trigger  $Ferr$  and in which the signal  $ack$  is never raised. In a constrained trigger query, the user provides, in addition to the system  $M$  and the behavior  $\varphi$ , also a regular expression  $c$  serving as a constraint for the possible solutions. In the above example,  $c = (\neg ack)^*$ . The solution for a constrained trigger query is the set of words that trigger  $\varphi$  in  $M$  and satisfy the constraint  $c$ .

Another variant is that of *observable trigger querying*. In many cases, the user would like to get a solution that depends only on a subset of the atomic propositions. For example, the user may wonder whether the environment can control the input signal  $req$  in a way that triggers the signal  $err$ , and if so, how. Technically, this corresponds to asking whether there is a word  $w$  over the alphabet  $2^{\{req\}}$  such that all words over  $2^{AP}$  that agree with  $w$  on the assignment to  $req$  trigger  $err$ . Thus, in addition to  $M$  and  $\varphi$ , the input to dominant trigger querying contains a set  $O \subseteq AP$  of observable atomic propositions, and the solution is a set of words over  $2^O$ .

The last variant of trigger querying we consider (in fact, it is more dual than variant) is the problem of finding *necessary conditions*. Recall that a word  $w$  triggers a behavior  $\varphi$  in  $M$  if all the computations of  $M$  with prefix  $w$  continue to a suffix that satisfies  $\varphi$ . Thus, a word triggering  $\varphi$  can be viewed as a sufficient condition for  $\varphi$  to happen in  $M$ , and trigger querying can be viewed as the problem of finding the set of sufficient conditions. Dually, a set of necessary conditions for  $\varphi$  to happen in  $M$  is a set  $N \subseteq (2^{AP})^*$  such that for every computation  $\pi$  of  $M$  and position  $i > 0$ , if the suffix of  $\pi$  from position  $i$  satisfies  $\varphi$ , then the prefix of  $\pi$  up to position  $i$  is in  $N$ . As with traditional query checking,  $\varphi$  may have several sets of necessary conditions, and we are interested in the strongest one, where strongest here means minimal in the language-containment partial order. Unlike traditional query checking, we show that a unique strongest necessary condition always exists. We also show that finding necessary conditions is computationally easier than finding sufficient solutions (i.e., trigger querying), and is only polynomial in the size of  $M$ .

## II. PRELIMINARIES

A word over an alphabet  $\Sigma$  is a sequence of letters from  $\Sigma$ . A word may be finite or infinite. We denote by  $\Sigma^*$  ( $\Sigma^\omega$ ) the set of finite (resp. infinite) words over  $\Sigma$ . Also,  $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ . For an infinite word  $w = w_0w_1w_2\dots$  and natural numbers  $i \leq j$  we denote by  $w[i..j]$  the finite word  $w_iw_{i+1}\dots w_j$  and denote by  $w^i$  the infinite word  $w_iw_{i+1}\dots$ .

A language is a set of words. For a language  $L \subseteq \Sigma^\infty$ , we denote by  $\text{pref}(L)$  the set of prefixes of words in  $L$ . That is,  $\text{pref}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^\infty \text{ such that } uv \in L\}$ . For a finite word  $w \in \Sigma^*$  and a regular expression  $r$  over  $\Sigma$ , we use  $w \models r$  to indicate that  $w \in L(r)$ .

A Kripke structure is a quintuple  $M = \langle AP, Q, Q_0, R, L \rangle$ , where  $AP$  is a set of atomic propositions,  $Q$  is a set of states,  $Q_0 \subseteq Q$  is a set of initial states,  $R \subseteq Q \times Q$  is a transition relation, and  $L : Q \rightarrow 2^{AP}$  is a labelling function that labels each state with the set of atomic propositions that hold in it. We assume that the transition relation is total; i.e., for every state  $q$  there exists at least one state  $q'$  such that  $R(q, q')$ . A sequence of states  $q_0, q_1 \dots$  is a *computation* of  $M$  if  $q_0 \in Q_0$  and for every  $i \geq 0$ , we have  $R(q_i, q_{i+1})$ . Unless we note otherwise, the computation is infinite. Each computation  $q_0, q_1 \dots$  induces the word  $L(q_0)L(q_1)\dots$  over the alphabet  $2^{AP}$ . The set of words induced by computations of  $M$  is called the *language of  $M$*  and is denoted by  $L(M)$ . Note that  $L(M) \subseteq (2^{AP})^\omega$ .

For an LTL formula  $\varphi$ , a Kripke structure  $M$ , and a set of states  $S$ , we use  $M, S \models \varphi$  to indicate that all the states in  $S$  satisfy  $\varphi$ . That is, all the computations that start in states in  $S$  satisfy  $\varphi$ . When  $S = S_0$ , we write  $M \models \varphi$ . Also, when  $S = \{s\}$  is a singleton, we write  $M, s \models \varphi$ .

The specification language PSL [9] introduces the *suffix-implication* operator, denoted  $\mapsto$ . Similar operators exist in other modern specification formalisms such as the operator TRIGGERS in ForSpec [1] and in SVA (in fact, in SVA the trigger operator is the only temporal operator) [17]. The syntax of suffix implication is as follows. Let  $AP$  be the set of atomic propositions. For a regular expression  $r$  over the alphabet  $2^{AP}$  and an LTL formula  $\varphi$  over  $AP$ , the expression  $r \mapsto \varphi$  is in PSL.<sup>3</sup> The semantics of suffix implication is that an infinite word  $w \in (2^{AP})^\omega$  satisfies  $r \mapsto \varphi$  iff for every  $i \geq 0$ , if  $w[0..i] \models r$  then  $w^i \models \varphi$ . Note the overlap in the  $i$ -th letter, which appears both in  $w[0..i]$  and in  $w^i$ . Note also that the semantics ignores empty prefixes of  $w$ . For a Kripke structure  $M$ , a regular expression  $r$ , and an LTL formula  $\varphi$ , we say that  $r$  triggers  $\varphi$  in  $M$ , denoted  $M \models r \mapsto \varphi$ , if all words in  $L(M)$  satisfy  $r \mapsto \varphi$ .

## III. TRIGGER QUERYING

For a Kripke structure  $M$  and an LTL formula  $\varphi$ , trigger querying deals with question of the type “for which words  $w \in \Sigma^*$ , it holds that  $M \models w \mapsto \varphi$ ”. We denote this instance of trigger querying by  $M \models? \mapsto \varphi$ . The *solution*

<sup>3</sup>In PSL, the regular expression  $r$  is not defined directly over the alphabet  $2^{AP}$ , but rather over the alphabet of Boolean expressions over  $2^{AP}$ . In our setting,  $r$  would be the output of trigger querying, and it is natural to return it as a regular expression over the alphabet  $2^{AP}$ .

of the trigger query  $M \models? \mapsto \varphi$  is the language of words that trigger  $\varphi$  in  $M$ ; i.e.  $L = \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$ . While the solution language  $L$  may be infinite, the finiteness of  $M$  implies that  $L$  is always regular. We therefore restrict our attention to regular expressions (or finite automata) as representations of the solution language. Thus, a solution to the trigger query  $M \models? \mapsto \varphi$  is a regular expression  $r$  for which  $L(r) = \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$ . Note that the solution  $r$  is the maximal (in terms of language containment) regular expression that satisfies  $M \models r \mapsto \varphi$ .

*Remark 1:* The definition of trigger querying is not as useful as it first seems because of a technical subtlety: A word  $w \in \Sigma^*$  that is not a finite computation of  $M$  is a *vacuous solution* to the trigger query  $M \models? \mapsto \varphi$  for any formula  $\varphi$ . Vacuous solutions are rarely interesting to users and are still members in the solution we define. In Section V-A, we define *relevant trigger querying*, which excludes such solutions. As discussed there, relevant trigger querying is technically very similar to trigger querying, in the sense that the algorithms and results about one variant carry on to the other one with minor changes. ■

As a first step toward solving trigger querying, we provide an alternative characterization for the set of words  $w$  that satisfy  $M \models w \mapsto \varphi$ . Consider a word  $w = w_0 \dots w_n \in \Sigma^*$ , and a finite computation  $s = s_0 \dots, s_n$  of  $M$ . We say that  $s$  *induces*  $w$  iff for every  $i \in \{0, \dots, n\}$ , it holds that  $L(s_i) = w_i$ . Note that a word  $w$  may be induced by several finite computations. We denote the set of finite computations that induce  $w$  by  $\text{induce}(w)$ . Also, we denote by  $\delta(w)$  the set of states  $s$  for which there exists a finite computation ending in  $s$  that induces  $w$ . Formally,  $\delta(w) = \{s_n \in Q \mid \exists s_0 \dots s_n \in \text{induce}(w)\}$ .

*Lemma 2:* For a Kripke structure  $M$  and an LTL formula  $\varphi$ , it holds that  $M \models w \mapsto \varphi$  iff  $M, \delta(w) \models \varphi$ .

*Proof:* Assume first that  $M, \delta(w) \models \varphi$ . Thus, every computation starting in a state in  $\delta(w)$  satisfies  $\varphi$ . Therefore, since every prefix of a computation of  $M$  that induces  $w$  ends in  $\delta(w)$ , we get that  $M \models w \mapsto \varphi$ .

For the other direction, assume that  $M \models w \mapsto \varphi$ . If  $\delta(w) = \emptyset$  then  $M, \emptyset \models \varphi$  vacuously. Otherwise, consider a state  $s \in \delta(w)$ . We show that  $M, s \models \varphi$ . Since  $s \in \delta(w)$ , there exists a finite computation  $\pi = s_0s_1\dots s_n$  of  $M$  such that  $s = s_n$  and  $\pi$  induces  $w$ . Assume, by way of contradiction, that  $M, s \not\models \varphi$ . Then, there exists a computation  $s, s^1s^2\dots$  that does not satisfy  $\varphi$ . Consider the computation  $c = s_0s_1\dots s_{n-1}ss^1s^2\dots$  of  $M$ . By the above,  $c \not\models w \mapsto \varphi$ , contradicting the assumption that  $M \models w \mapsto \varphi$ . ■

The alternative characterization allows us to reduce trigger querying to global model checking of  $\varphi$  on  $M$ .

*Theorem 3:* Trigger querying can be solved in polynomial space, and is PSPACE-hard.<sup>4</sup>

*Proof:* Let  $M = \langle AP, Q, Q_0, R, L \rangle$  and let  $\llbracket \varphi \rrbracket_M = \{s \in Q \mid M, s \models \varphi\}$  denote the set of states  $s$  for which  $M, s \models \varphi$ . Computing  $\llbracket \varphi \rrbracket_M$  is the global model-checking problem for LTL, and is known to be in PSPACE [16].

<sup>4</sup>Trigger querying is not a decision problem and therefore cannot be PSPACE-complete.

By Lemma 2, the solution to the trigger query  $M \models? \mapsto \varphi$  is the language  $L = \{w \in \Sigma^* \mid \delta(w) \subseteq \llbracket \varphi \rrbracket_M\}$ . We construct a deterministic finite automaton  $\mathcal{A}$  such that  $L(\mathcal{A}) = L$ . Finding a regular expression equivalent to  $\mathcal{A}$  can be done using standard methods.

Intuitively, we would like to “transform  $M$  into an automaton” and then apply the subset construction setting the accepting states to be nonempty subsets of  $\llbracket \varphi \rrbracket_M$ . In an automaton, the alphabet is on the transitions, whereas in a Kripke structure, it is on the states. We move the label of a state to the transitions into the state, which makes it easier to deal with the overlap between the prefix and the suffix in the definition of the trigger operator.

We define  $\mathcal{A} = \langle \Sigma, 2^Q \cup \{q_{in}\}, \{q_{in}\}, \rho, F \rangle$ , where  $q_{in}$  is a new state, and  $\rho$  and  $F$  are defined as follows:

- The transition relation  $\rho$  is defined for every  $\sigma \in 2^{AP}$  as follows. First, for the state  $q_{in}$ , we define  $\rho(q_{in}, \sigma) = \{s \mid s \in Q_0 \text{ and } L(s) = \sigma\}$ . Then, for a state  $S \in 2^Q$ , we define  $\rho(S, \sigma) = \bigcup_{q \in S} \{s \mid R(q, s) \text{ and } L(s) = \sigma\}$ .
- The set  $F$  of accepting states is the collection of subsets of  $\llbracket \varphi \rrbracket_M$ . Thus,  $F = \{S \subseteq Q \mid S \subseteq \llbracket \varphi \rrbracket_M\}$ .

It is not hard to prove by an induction on the length of  $w$  that  $L(\mathcal{A})$  contains only words  $w$  such that  $\delta(w) \subseteq \llbracket \varphi \rrbracket_M$ . Hence, by Lemma 2,  $L(\mathcal{A})$  is the solution to  $M \models? \mapsto \varphi$ .

We turn now to the lower bound. PSPACE-hardness follows from the PSPACE hardness of LTL model checking. Indeed, for every Kripke structure  $M$  and LTL formula  $\varphi$ , we have that  $M \models \varphi$  iff the solution to the trigger query  $M \models? \mapsto \varphi$  contains  $2^{AP}$  (that is, all words of length 1). Unfortunately, the situation for trigger querying is worse as the complexity of the upper bound above is polynomial space in the size of  $M$  and not only in the length of  $\varphi$ . Accordingly, we now prove that the *structure complexity* of trigger querying, that is, the complexity in terms of the Kripke structure, assuming the formula is of a fixed length, is PSPACE-hard.

The proof is by a reduction from universality of non-deterministic automata on finite words (NFA, for short), which is known to be PSPACE-hard [14]. Given an NFA  $\mathcal{A}$ , we construct a Kripke structure  $M$  and a formula  $\varphi$  of a fixed length, such that for every word  $w \in \Sigma^*$ , it holds that  $M \models w \mapsto \varphi$  iff  $w \notin L(\mathcal{A})$ . Thus, a solution to the trigger query  $M \models? \mapsto \varphi$  is an automaton that complements the language of  $\mathcal{A}$ . Since the length of  $\varphi$  is fixed, the PSPACE-hardness that follows is indeed for the structure complexity. Let  $\mathcal{A} = \langle 2^{AP}, Q, Q_0, \rho, F \rangle$ . To avoid vacuous solutions, we assume that  $\mathcal{A}$  has a run on every finite word (if this is not the case, we can add a rejecting sink). Intuitively, we introduce a new atomic proposition  $a$ , and construct  $M$  such that for every word  $w \in \Sigma^*$ , the word  $w \cdot a^\omega \in L(M)$  iff  $w \in L(\mathcal{A})$ . Thus,  $M \models w \mapsto X \neg a$  iff  $w \notin L(\mathcal{A})$ .

In the NFA  $\mathcal{A}$ , the alphabet is on the transitions, whereas in the Kripke structure  $M$ , the alphabet is on the states. We construct  $M$  such that each state of it is associated with a state of  $\mathcal{A}$  and the letter  $\mathcal{A}$  is about to read. Thus, a run  $q_0 q_1 \dots q_n$  of  $\mathcal{A}$  on  $w_1 w_2 \dots w_n$  corresponds to the computation  $\langle q_0, w_1 \rangle \langle q_1, w_2 \rangle \dots \langle q_{n-1}, w_n \rangle$  of  $M$ . In addition to the states associated with  $\mathcal{A}$ 's states and letters,  $M$  contains

a special state  $q_a$ , labelled  $\{a\}$ , corresponding to acceptance in  $\mathcal{A}$ . Formally,  $M = \langle AP \cup \{a\}, S, S_0, R, L \rangle$ , where

- $S = (Q \times \Sigma) \cup \{q_a\}$  where  $q_a$  is a new state.
- $S_0 = Q_0 \times \Sigma$ .
- $R = \{(\langle q, \sigma \rangle, \langle q', \sigma' \rangle) \mid q' \in \rho(q, \sigma)\} \cup \{(\langle q, \sigma \rangle, q_a) \mid \rho(q, \sigma) \cap F \neq \emptyset\} \cup \{(q_a, q_a)\}$ .
- $L(\langle q, \sigma \rangle) = \sigma$  and  $L(q_a) = \{a\}$ .

For  $n > 0$ , it is not hard to prove, by an induction on  $n$ , that  $\langle q_0, w_1 \rangle \langle q_1, w_2 \rangle \dots \langle q_{n-1}, w_n \rangle \in \text{pref}(L(M))$  iff  $q_0 q_1 \dots q_{n-1}$  is a run of  $\mathcal{A}$  on  $w_1 \dots w_{n-1}$ . Therefore,  $w \cdot a^\omega \in L(M)$  iff  $w \in L(\mathcal{A})$ . It follows that  $M \models w \mapsto X \neg a$  iff  $w \notin L(\mathcal{A})$  as desired. ■

*Remark 4:* A note for readers familiar with alternating automata: Modulo the technical issue of the alphabet being on the transitions vs. the states, the automaton  $\mathcal{A}$  is simply  $M$  when viewed as a universal automaton with  $\llbracket \varphi \rrbracket_M$  being the set of accepting states. The construction described in the proof translates this automaton to a deterministic one. ■

#### IV. PARTIAL SOLUTIONS FOR TRIGGER QUERYING

As shown in Section III, the complexity of solving trigger querying is polynomial space in the size of the system. Unfortunately, such complexity might prove infeasible for many practical systems. Therefore, practical considerations lead us to search for partial yet more efficient solutions.

A reasonable approach is to search for subsets of the solution to a trigger query. The motivation for such an approach is that a user that is unable to get a complete characterisation of the words that trigger a behavior is usually still interested in specific scenarios that trigger the behavior.

A partial solution to a trigger query  $M \models? \mapsto \varphi$  is a subset of the solution to the trigger query. We allow the subset to be empty only if the complete solution is empty. Also, as in the case of complete solutions, we restrict attention to regular subsets. Formally, a partial solution to the trigger query  $M \models? \mapsto \varphi$  is a regular expression  $r$  such that  $L(r) \subseteq \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$  and  $L(r) = \emptyset$  iff  $\{w \in \Sigma^* \mid M \models w \mapsto \varphi\} = \emptyset$ .

The search of partial solutions may take various forms. A natural possibility is to search for a single word as a partial solution to a trigger query. It follows from the proof of Theorem 3, however, that the system complexity of deciding whether the solution to a trigger query is not empty is already PSPACE-hard. We therefore move to the next natural possibility, which is to search for a single word of a given bounded length. If such a bound is given in binary, however, the structure complexity of the problem remains PSPACE-hard.<sup>5</sup> An algorithm that searches for a single word in the solution is interesting if it also outputs the word. It seems natural, therefore, to consider the case in which the length bound is given in unary. Accordingly, partial trigger querying gets as input a structure  $M$ , a fixed formula  $\varphi$ , and a length

<sup>5</sup>This follows from the hardness proof in Theorem 3. There, given an NFA  $\mathcal{A}$ , we can reduce the problem of deciding the universality of  $\mathcal{A}$  to trigger querying. It is not hard to see that  $\mathcal{A}$  is universal iff it accepts all words of length exponential in its size. Therefore, if we allow exponential bounds (whose binary encoding is polynomial), the hardness proof carries over to the partial-solution case.

bound  $n$ , given in unary, and decides whether there exists a word  $w$  of length at most  $n$  such that  $M \models w \mapsto \varphi$ .

*Theorem 5: Partial trigger querying is NP-complete.*

*Proof:* For the upper bound, we show that we can check, given a word  $w$ , whether  $M \models w \mapsto \varphi$  in time polynomial in  $M$  and the length of  $w$  (although not necessarily polynomial in  $\varphi$ ). Recall that  $M \models w \mapsto \varphi$  iff  $\delta(w) \subseteq \llbracket \varphi \rrbracket_M$ . Computing both  $\llbracket \varphi \rrbracket_M$  and  $\delta(w)$  can be done in time polynomial in  $M$  and the length of  $w$ . Thus, the decision problem is in NP.

We proceed to prove the lower bound by a reduction from the following NP-complete problem [12]: given a directed graph  $G = (V, E)$  and two vertices  $s, t \in V$ , are there two vertex-disjoint paths, one from  $s$  to  $t$  and one from  $t$  to  $s$  (the only vertices that appear in both paths are  $s$  and  $t$ ).

For clarity of the presentation, we first present the reduction to relevant trigger querying, in which the solution contains only non-vacuous triggers (see Remark 1). We will later comment on the technical adjustment to the general case.

Consider a graph  $G = (V, E)$ , and two vertices  $s, t \in V$ . Let  $n = |V|$ . Note that if two vertex-disjoint paths from  $s$  to  $t$  and back exist, then two such paths of length at most  $n$  exist. We therefore restrict our attention to paths of length at most  $n$ . To simplify things further, we add an edge from  $t$  to itself (if it does not exist). Then, we can also assume that the paths are of the same length, as otherwise we can pad the shorter path with  $t$ 's. Finally, we assume that each state in  $G$  has a successor (otherwise, we can add a sink to which all dead-ends go).

We intend to construct a Kripke structure  $M$  and a formula  $\varphi$  such that the solution to  $M \models? \mapsto \varphi$  contains a word of length at most  $n$  if there exist two vertex-disjoint paths from  $s$  to  $t$  and back, and is empty otherwise. Let  $\Sigma = (V \times V) \cup \{p\}$ . For simplicity, we assume that the atomic propositions of  $M$  encode letters in  $\Sigma$ . A finite computation of  $M$  that does not reach a state labelled  $p$  encode two sequences of vertices in  $G$ . For example, the word  $(x_1, y_1) \dots (x_k, y_k)$  encodes the two sequences of vertices  $x_1, \dots, x_k$  and  $y_1, \dots, y_k$ . We define the transitions of  $M$  so that the projection of a finite computation on the first element encodes a path from  $s$  in  $G$ , while the projection on the second element encodes a path from  $t$ . For both elements the path may be followed by a  $p^*$  suffix. In addition, the following would hold.

- 1) If  $w \in \Sigma^*$  encodes two vertex-disjoint paths from  $s$  to  $t$  and back, then for all infinite suffixes  $v$  for which  $wv \in L(M)$ , the first letter of  $v$  is  $p$ .
- 2) If  $w \in \Sigma^*$  does not encode two vertex-disjoint paths from  $s$  to  $t$  and back, then there exists an infinite suffix  $v$  whose first letter is not  $p$  and  $wv \in L(M)$ .

If we succeed in constructing  $M$  as above, then for every word  $w \in \Sigma^*$ , we have that  $w$  is a non-vacuous solution to  $M \models? \mapsto Xp$  iff  $w$  encodes two vertex-disjoint paths from  $s$  to  $t$  and back. Thus, setting  $\varphi = Xp$ , we are done.

We now proceed to define  $M$  in detail. In order to know whether the two paths that correspond to a finite computation of  $M$  are vertex-disjoint,  $M$  chooses nondeterministically one vertex from each path and records it. If the first path reaches  $t$  and the second path reaches  $s$ , then  $M$  compares the recorded

vertices. If the recorded vertices differ,  $M$  enters a sink state labelled  $p$ . If, on the other hand, the recorded vertices are the same,  $M$  continues to visit states that are not labelled by  $p$ . Note that when the paths are vertex-disjoint, the recorded vertices must be different. On the other hand, when the two paths are not vertex-disjoint, and share a vertex  $v$ , then there is a computation of  $M$  that generates these paths and chooses to record the vertex  $v$  for both paths.

The states of  $M$  are tuples in  $V \times V \times (V \cup \{\perp\}) \times (V \cup \{\perp\})$  (as well as the special sink state  $s_p$ ). In the state  $\langle v_1, v_2, x_1, x_2 \rangle$ , the values  $v_1$  and  $v_2$  stand for the current vertices in the first and second paths, and the values  $x_1$  and  $x_2$  stand for the recorded vertices from these paths. The symbol  $\perp$  stands for “no vertex is recorded yet”. A state  $\langle v_1, v_2, x_1, x_2 \rangle$  is labelled by atomic propositions encoding the letter  $\langle v_1, v_2 \rangle$ . The state  $s_p$  is used to generate the  $p^*$  suffixes and is labelled by  $p$ . The state  $\langle s, t, \perp, \perp \rangle$  is the single initial state. The transition relation makes sure that a computation of  $M$  generates only sequences of pairs that correspond to paths in  $G$ . In addition, if the third (resp. fourth) element of a state is  $\perp$ , then a nondeterministic choice is made whether to record the current vertex  $v$  of the first (resp. second) path, which is done by replacing  $\perp$  with  $v$ . If the third (resp. fourth) element is not  $\perp$ , then it retains its value. Finally, if a vertex of the type  $\langle t, s, x_1, x_2 \rangle$  is reached and  $x_1 \neq x_2$  (or  $x_1$  equals  $x_2$  but both equal  $s, t$ , or  $\perp$ ) then a transition to  $s_p$  is taken.

Formally,  $M = \langle AP, S, S_0, R, L \rangle$ , where  $AP, S, S_0$ , and  $L$  are defined above. Before defining  $R$ , we introduce the following notation. For  $v \in V$  and  $x \in V \cup \{\perp\}$ , we denote by  $rec(v, x)$  the set  $\{v, \perp\}$  if  $x = \perp$  and the set  $\{x\}$  if  $x \neq \perp$ . We proceed to define  $R$  (we define it as a function  $R : S \rightarrow 2^S$ ):

- $R(s_p) = \{s_p\}$ .
- For states of the type  $\langle t, s, x_1, x_2 \rangle$  in which  $x_1 \neq x_2$ , or  $x_1 = x_2$  but  $x_1 \in \{s, t, \perp\}$ , set  $R(\langle t, s, x_1, x_2 \rangle) = \{s_p\}$ .
- For all other states,  $R(\langle v_1, v_2, x_1, x_2 \rangle) = \{ \langle v'_1, v'_2, x'_1, x'_2 \rangle \mid E(v_1, v'_1), E(v_2, v'_2), x'_1 \in rec(v_1, x_1), x'_2 \in rec(v_2, x_2) \}$ .

It is not hard to prove that for every word  $w \in \Sigma^*$ , we have that  $w$  is a non-vacuous solution to  $M \models? \mapsto Xp$  iff  $w$  encodes two vertex-disjoint paths from  $s$  to  $t$  and back. Note that the reduction is polynomial in the size of  $G$ , and that  $\varphi$  is fixed.

Since words  $w$  in the solution to  $M \models? \mapsto Xp$  may be vacuous solutions, the reduction shows that relevant trigger query is NP-hard. In order to prove NP-hardness for trigger querying, we modify  $M$  so that  $M \models? \mapsto Xp$  would not have vacuous solutions. For that, we have to modify  $M$  to a Kripke structure  $M'$  such that  $pref(M') = \Sigma^*$ . We should make sure that the non-vacuous solutions to  $M \models? \mapsto Xp$  continue to trigger  $Xp$  in  $M'$ . Thus, reading such a solution, we must move to  $s_p$ . We do this by defining  $M'$  to subsume  $M$  in such a way that for every word  $w \in \Sigma^*$ , if  $w \in L(M)$ , then the set of computations that induce  $w$  in  $M'$  is equal to the set of computations that induce it in  $M$ . Thus,  $M'$  only generates new words but does not add ways to generate words that are already in  $L(M)$ . It is not hard to define  $M'$  by adding to  $M$  a component that generate  $\Sigma^*$  and to which computations get whenever they get stuck in  $M$ . ■

### A. Practical considerations

We suggest two symbolic methods for searching for partial solutions to a trigger query. Both methods consider a bound  $n > 0$  on the length of words in the solution. First, a BDD-based method that computes all words of length  $n$  in the solution. Second, a SAT-based method that searches for a single word of length  $n$  in the solution.<sup>6</sup> For reasons explained below, we recommend the BDD-based method.

Let  $\Sigma = 2^{AP}$ . The main task of the BDD procedure is to compute the set  $\{\langle w_1, \dots, w_n, s \rangle \in \Sigma^n \times S \mid s \in \delta(w_1 \dots w_n)\}$ . For this purpose, we need BDD variables to represent letters and states.<sup>7</sup> We encode the transition relation as a set  $R \subseteq S \times \Sigma \times S$  where  $\langle s, \sigma, s' \rangle \in R$  if  $s'$  is a successor of  $s$  and  $L(s') = \sigma$ .

We use two vectors of BDD variables  $\vec{s}$  and  $\vec{s}'$ , encoding states. Intuitively,  $\vec{s}$  encodes current states and  $\vec{s}'$  successor states. In addition, we use  $n$  vectors of BDD variables  $\vec{w}_1, \dots, \vec{w}_n$  encoding letters. We also use another vector of BDD variables  $\vec{\sigma}$  encoding letters. In fact, the variables in  $\vec{\sigma}$  are not necessary but the presentation is clearer with  $\vec{\sigma}$ . We assume that the algorithm has access to BDDs for the set  $\llbracket \varphi \rrbracket_M(\vec{s})$ , the initial set  $S_0(\vec{s})$ , and the transition relation  $R(\vec{s}, \vec{\sigma}, s')$ . We also need a BDD  $B(\vec{s}, \vec{\sigma})$  for the set  $\{\langle s, L(s) \rangle \mid s \in S\}$ .

The function UNTAG gets a BDD with  $\vec{s}'$  variables and no  $\vec{s}$  variables and replaces all the  $\vec{s}'$  variables with the corresponding  $\vec{s}$  variables. Similarly, for each  $i \in \{1, \dots, n\}$ , the function  $i$ -TAG gets a BDD with  $\vec{\sigma}$  variables and no  $\vec{w}_i$  variables and replaces all the  $\vec{\sigma}$  variables with the corresponding  $\vec{w}_i$  variables.

We describe the algorithm in Figure IV-A below.

---

#### Algorithm 1: BDD based algorithm

---

```

1  $X \leftarrow S_0$ ;
2  $X \leftarrow X \cap 1\text{-TAG}(B)$ ;
3 for  $i = 2$  to  $n$  do
4    $X \leftarrow \text{UNTAG}(\exists s X \cap i\text{-TAG}(R))$ ;
5 end
6  $Y \leftarrow \exists s (X \cap \neg \llbracket \varphi \rrbracket_M)$ ;
7  $Z \leftarrow \neg Y$ ;
8 return  $Z$ ;

```

---

Intuitively, in lines 1 – 5, the algorithm computes, in the BDD  $X$ , the set  $\{\langle \vec{w}_1, \dots, \vec{w}_i, \vec{s} \rangle \mid \vec{s} \in \delta(\vec{w}_1, \dots, \vec{w}_i)\}$ , for the  $i$ 's between 1 to  $n$ . Thus, after line 5, the BDD  $X$  contains exactly all tuples  $\{\langle w_1, \dots, w_n, s \rangle \mid s \in \delta(w_1 \dots w_n)\}$ .

Accordingly, in line 6, the algorithm computes all the words  $w_1 \dots w_n$  for which  $\delta(w_1 \dots w_n) \not\subseteq \llbracket \varphi \rrbracket_M$ , namely words that do not trigger  $\varphi$ . Finally, in line 7, the latter set is complemented resulting in the set of all words that do trigger  $\varphi$ , i.e., the solution to  $M \models? \vdash \varphi$ .

As the NP-complete complexity for partial trigger querying suggests, it is also possible to apply a SAT solver in order

<sup>6</sup>It is not hard to adapt the algorithms to words of length at most  $n$ . We present the versions for length exactly  $n$  since they are technically simpler.

<sup>7</sup>In real applications, both states and letters are subsets of the atomic propositions (an assignment to the atomic propositions induces a state, labelled by the letter that corresponds to the observable atomic propositions that are valid in the state). Our solution is general and does not assume such a relation between letters and states.

to find partial solutions. The formula to be considered can be built along the following lines: Let  $\vec{X}$  be a vector of variables representing a *set* of states of  $M$ . Let  $\vec{X}'$  be another such vector, and let  $\vec{\sigma}$  be a vector of variables representing a letter. We denote by  $\psi_R(\vec{X}, \vec{\sigma}, \vec{X}')$  a formula that is true iff  $\vec{X}'$  represents the set of states that are successors of a state in  $\vec{X}$  and whose labelling is  $\vec{\sigma}$ . Formally,  $\vec{X}' = \{q' \in S \mid \exists q \in \vec{X} \text{ such that } R(q, q') \text{ and } L(q') = \vec{\sigma}\}$ . Let  $\psi_\varphi(\vec{X})$  be a formula that is true iff  $\vec{X}$  represents a set that is contained in  $\llbracket \varphi \rrbracket_M$ . Finally, let  $\psi_I(\vec{X})$  be a formula that is true iff  $\vec{X}$  represents the set  $S_0$ . The formula to be fed into the SAT solver is  $\psi_I(\vec{X}_0) \wedge \bigwedge_{i=1}^n \psi_R(\vec{X}_{i-1}, \vec{w}_i, \vec{X}_i) \wedge \psi_\varphi(\vec{X}_n)$ , where  $\vec{X}_0, \dots, \vec{X}_n$  and  $\vec{w}_1, \dots, \vec{w}_n$  are (vectors of) free variables.

A satisfying assignment assigns values to the (vectors of) variables  $\vec{w}_1 \dots \vec{w}_n$  and  $\vec{X}_0, \dots, \vec{X}_n$ . It is not hard to see that the values assigned to  $\vec{w}_1 \dots \vec{w}_n$  encode a word that is partial solution for the trigger query, and that the values assigned to  $\vec{X}_0, \dots, \vec{X}_n$  encode the sets  $\delta(\vec{w}_1), \delta(\vec{w}_1 \vec{w}_2), \dots, \delta(\vec{w}_1 \dots \vec{w}_n)$ .

Note that unlike the case in bounded model checking, the suggested algorithm uses as many variables as are states in the structure  $M$  (rather than in the symbolic representation of  $M$ ). The technical need for so many variables arise from the need to consider *all* the elements of a set of states (encoded in the  $\vec{X}_i$ 's), and it occurs in other (already well challenged) contexts of bounded model checking, e.g., when evaluating the diameter of a model.

## V. VARIANTS OF TRIGGER QUERYING

In this section we present several natural variants of trigger querying.

### A. Relevant trigger querying

As noted in Remark 1, the definition of trigger querying allows the solution to contain vacuous solutions, namely words that are not induced by finite computations of  $M$ . Vacuous solutions are rarely interesting to users. In this section we define *relevant trigger querying*, which excludes vacuous solutions. We show that the algorithms and results we describe for trigger querying apply, with minor modifications, to the relevant case.

For a Kripke structure  $M$ , a word  $w \in \Sigma^*$ , and an LTL formula  $\varphi$ , the word  $w$  *relevantly triggers*  $\varphi$  in  $M$ , denoted  $M \models w \overset{r}{\vdash} \varphi$ , if  $w$  triggers  $\varphi$  in  $M$  and  $w$  is induced by a finite computation of  $M$ . The solution to the *relevant trigger query*  $M \models? \overset{r}{\vdash} \varphi$  is the set of words that relevantly trigger  $\varphi$  in  $M$  (i.e.,  $\{w \in \Sigma^* \mid M \models w \overset{r}{\vdash} \varphi\}$ ).

*Remark 6:* Note that our notion of vacuity does not coincide with the notion of vacuity in the context of model checking of trigger formulas [5]: when  $\llbracket \varphi \rrbracket_M = Q$  (for example, when  $\varphi = \mathbf{true}$ ), the regular expression  $r$  does not affect the satisfaction of  $r \vdash \varphi$  in  $M$ . In such cases, all finite computations of  $M$  are non-vacuous solutions according to our definition. It is easy to adjust our solutions to a definition that would cause all solutions to be vacuous in such cases. ■

Solving relevant trigger querying is very similar to solving trigger querying. It is not hard to see that a word  $w \in \Sigma^*$  is

induced by a finite computation of a Kripke structure  $M$ , iff  $\delta(w) \neq \emptyset$ . Thus, the “relevant counterpart” of Lemma 2 is as follows.

*Lemma 7: For a Kripke structure  $M$  and an LTL formula  $\varphi$ , it holds that  $M \models w \xrightarrow{r} \varphi$  iff  $\delta(w) \neq \emptyset$  and  $M, \delta(w) \models \varphi$ .*

It follows that the construction of  $\mathcal{A}$  in the proof of Theorem 3 is valid also for the case of relevant trigger querying, except that we have to remove the empty set from  $F$ . The lower bound proofs are also easy to adjust for the relevant case. Hence, we can conclude with the following.

*Theorem 8: Trigger querying can be solved in polynomial space, and is PSPACE-hard.*

### B. Constrained trigger querying

Trigger querying typically does not consist of a single query but rather it is an interactive dialog between the user and the trigger-query tool. A natural course of events is one in which the user refines the trigger queries in order to find scenarios that not only trigger the behavior in question, but also satisfy some constraints. For example, the user may search for scenarios that trigger  $Ferr$  and in which the signal  $ack$  is never raised. In a *constrained trigger query*, the user provides, in addition to the system  $M$  and the behavior  $\varphi$ , also a regular expression  $c$  serving as a mask for the possible solutions. In the above example,  $c = (\neg ack)^*$ . The solution for a constrained trigger query is the set of words that trigger  $\varphi$  in  $M$  and satisfy the constraint  $c$ .

Trigger querying can be viewed as a special case of constrained trigger querying with  $c = \mathbf{true}$ . Also, solving a constrained trigger query can proceed by solving the trigger query and intersecting the solution with the constraint language  $L(c)$  (the intersection can be implemented as intersection of finite automata). Hence, we have the following.

*Theorem 9: Constrained trigger querying can be solved in polynomial space and is PSPACE-hard.*

Constrained trigger querying is of special interest when combined with partial trigger querying. Note that in the unconstrained case, it is possible to solve the trigger query and only then intersect the solution with the constraint. In partial trigger querying, such a course of action may lead to an empty set of partial solutions although the set of solutions that satisfy the constraint is not empty. Therefore, the constraint must be taken into account during the search for partial solutions (rather than after it). In practice, it is not hard to modify the algorithms suggested in Subsection IV-A to take the constraint  $c$  into account while searching for a partial solution.

Note that relevant trigger querying can be viewed as a special case of constrained trigger querying — one in which the constraint is the set of words induced by a finite computation of the Kripke structure. Nevertheless, the direct algorithms for relevant trigger querying are simpler than these that follow from this view.

### C. Observable trigger querying

In many cases, the user would like to get a solution to a trigger query that depends only on a subset of the atomic propositions. For example, the user may wonder whether

the environment can control the input signal  $req$  in a way that triggers the signal  $err$ , and if so, how. Technically, this corresponds to asking whether there is a word  $w$  over the alphabet  $2^{\{req\}}$  such that all words over  $2^{AP}$  that agree with  $w$  on the assignment to  $req$  trigger  $err$ .

Formally, for a word  $w$  over  $2^{AP}$  and a set  $O \subseteq AP$ , let  $w|_O$  be the word over  $2^O$  obtained from  $w$  by projecting its letters on  $O$ . Then, for a word  $w' \in 2^O$ , let  $wide(w', AP \setminus O) = \{w : w|_O = w'\}$  be the set of words over  $2^{AP}$  obtained from  $w'$  by extending its letters to  $AP$ . The input to *observable trigger querying* contains, in addition to  $M$  and  $\varphi$ , also a set  $O \subseteq AP$  of observable atomic propositions. The solution to the observable trigger query  $M \models? \mapsto \varphi$  with a set  $O$  of observable atomic propositions is the set  $\{w' : M \models w \mapsto \varphi \text{ for all } w \in wide(w', AP \setminus O)\}$ .

Note that the observable atomic propositions are “observable to the query”. Thus, unlike the standard interpretation of observable and non-observable atomic propositions, here the idea is not to hide internal signals, but rather to restrict attention to the atomic propositions in terms of which we want the solution to the trigger query to be expressed. Often, these propositions would be related to implementation details or other internal signals that are considered non-observable in the standard context.

*Theorem 10: Observable trigger querying can be solved in polynomial space and is PSPACE-hard.*

*Proof:* For the upper bound, we modify the construction of the NFA  $\mathcal{A}$  described in the proof of Theorem 3 as follows. The modification is only in the definition of  $\rho$ , which now assumes the alphabet  $2^O$ . For a letter  $\sigma \in 2^O$ , we define  $\rho(q_{in}, \sigma) = \{s \mid s \in Q_0 \text{ and } L(s) \cap O = \sigma\}$ , and for a state  $S \in 2^Q$ , we define  $\rho(S, \sigma) = \bigcup_{q \in S} \{s \mid R(q, s) \text{ and } L(s) \cap O = \sigma\}$ . Thus, the states in the successor states have to agree with  $\sigma$  on the atomic propositions in  $O$ , and all other atomic propositions are ignored. It is not hard to prove that  $w \in 2^O$  is accepted by the modified NFA iff all the words in  $wide(w, AP \setminus O)$  are accepted by  $\mathcal{A}$ .

Since trigger querying can be viewed as a special case of observable trigger querying (with  $O = AP$ ), PSPACE-hardness follows from PSPACE-hardness of trigger querying. ■

*Remark 11:* A note for readers familiar with alternating automata and Remark 4. Recall that the automaton  $\mathcal{A}$  constructed in the proof of Theorem 3 is  $M$  when viewed as a universal automaton. For nondeterministic automata, the existential projection of an automaton over the alphabet  $\Sigma_1 \times \Sigma_2$  to an automaton over the alphabet  $\Sigma_1$  can be easily done by ignoring the  $\Sigma_2$  component in the transitions. For universal automata, existential projection involves an exponential blow up, but universal projection is easy. This is why the transition to observable trigger querying, which corresponds to a universal projection of the solution, does not make the problem more complex. ■

### D. Necessary conditions

Recall that a word  $w$  triggers a behavior  $\varphi$  in  $M$  if all the computations of  $M$  with prefix  $w$  continue to a suffix

that satisfies  $\varphi$ . Thus, a word triggering  $\varphi$  can be viewed as a sufficient condition for  $\varphi$  to happen in  $M$ , and trigger querying can be viewed as the problem of finding the set of sufficient conditions. In this section, we study the dual problem, namely finding the set of necessary conditions for  $\varphi$  to happen in  $M$ .

A *necessary condition* to  $\varphi$  in  $M$  is a regular expression  $r$  such that for every computation  $\pi$  of  $M$  and for every  $i \geq 0$ , if  $\pi^i \models \varphi$ , then  $\pi[0..i] \models r$ . It is easy to see that  $\varphi$  may have several necessary conditions in  $M$  (in fact,  $\Sigma^*$  is always a necessary condition). The necessary conditions, however, are (partially) ordered by language containment. We say that a necessary condition  $r$  is *stronger* than a necessary condition  $r'$ , if  $L(r) \subseteq L(r')$ .

We show that there always exists a unique strongest interesting necessary condition. Let  $G \subseteq S$  denote the set of states from which there is a computation that satisfies  $\varphi$ . Formally,  $G = \{s \in S \mid M, s \not\models \neg\varphi\}$ . Let  $r$  be a regular expression for the set of words  $w \in \Sigma^*$  for which there exists a finite computation of  $M$  that induces  $w$  and ends in  $G$ . Formally,  $L(r) = \{w \in \Sigma^* \mid \delta(w) \cap G \neq \emptyset\}$ .

For every computation  $\pi$  of  $M$  and for every  $i \geq 0$ , if  $\pi^i \models \varphi$ , then, by the definition of  $G$ , it must be that  $\pi[0..i] \in L(r)$ . Hence,  $r$  is a necessary condition. We prove that for every necessary condition  $r'$ , we have  $L(r) \subseteq L(r')$ , thus  $r$  is the strongest necessary condition. Let  $r'$  be a necessary condition. Consider a word  $w \in L(r)$ . Let  $\pi$  and  $i$  be such that  $w$  is induced by  $\pi[0..i]$ . Let  $s$  be the last state in  $\pi[0..i]$ . Since  $s$  is in  $G$ , there exists some computation  $\pi_s$  of  $M$  that starts in  $s$  and satisfies  $\varphi$ . The concatenation of  $\pi[0..i-1]$  and  $\pi_s$  is a computation of  $M$  whose suffix from position  $i$  satisfies  $\varphi$ . Since  $r'$  is a necessary condition, it must be that  $\pi[0..i] \in L(r')$ .

*Theorem 12: Finding the strongest necessary condition is PSPACE-complete. The structure complexity of the problem is nondeterministic logarithmic space.*

*Proof:* We start with the upper bounds. The set  $G$  above can be computed by solving the global LTL model-checking problem ( $G = Q \setminus \llbracket \neg\varphi \rrbracket_M$ ). An NFA for the strongest necessary condition can be obtained from  $M$  by moving the labels from the states to the transitions into the state (a new initial state should be added), and defining  $G$  as the set of accepting states. Since global LTL model-checking is in PSPACE and its structure complexity is NLOGSPACE, we are done. The lower bound also follows from LTL model checking: Checking whether  $M \models \varphi$  can be reduced to checking whether there is an initial state in the set  $\{s \in S \mid M, s \not\models \varphi\}$ , thus, it is reducible to finding the strongest necessary condition for  $\neg\varphi$ , in a Kripke structure that is obtained from  $M$  by marking the initial states with a special atomic proposition. ■

## VI. DISCUSSION

We introduced and studied trigger querying — a model-exploration problem in which one searches for the set of scenarios that trigger a behavior in a system.

Algorithms and modelling techniques originally developed for formal verification have turned out to be useful in other areas. This includes, for example, modelling and reasoning

about biological systems [11], [10], business processes [13], and AI plans [2]. We believe that the application of trigger querying in these areas is very natural. Indeed, the type of questions one cares about in these areas are of the form “which scenarios trigger an action of a particular cell / an activation of some item in a contract / an action of the robot’s arm.”

Finally, our work here can be viewed as a first step towards a general temporal-query checking methodology, in which the “?” place-holder may be replaced by a temporal behavior rather than a propositional assertion. It looks like the most appropriate replacement for “?” are temporal events of a bounded duration, and the most convenient way to specify them are regular expressions, as done here. Also, triggers seem to capture a large fraction of the natural model-exploration questions interesting in practice. This, together with the popularity of the triggers operator in industrial setting has convinced us to restrict attention to trigger querying. At any rate, the ideas introduced here for trigger querying are useful in the general case.

## REFERENCES

- [1] 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 logic. In *Proc. 8th TACAS*, LNCS 2280, pages 296–211. Springer, 2002.
- [2] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. of Mathematics and Artificial Intelligence*, 22:5–27, 1998.
- [3] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc 13th CAV*, LNCS 2102, pages 363–367. Springer, 2001.
- [4] G. Bruns and P. Godefroid. Temporal logic query checking. In *Proc. 16th LICS*, pages 409–420. IEEE Computer Society, 2001.
- [5] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, LNCS 3725, pages 191–206. Springer, 2005.
- [6] W. Chan. Temporal-logic queries. In *Proc 12th CAV*, LNCS 1855, pages 450–463. Springer, 2000.
- [7] M. Chechik, M. Gheorghiu, and A. Gurfinkel. Finding state solutions to temporal queries. In *Proc. Integrated Formal Methods*, 2007. To appear.
- [8] M. Chechik and A. Gurfinkel. TLQSolver: A temporal logic query checker. In *Proc 15th*, LNCS 2725, pages 210–214. Springer, 2003.
- [9] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.
- [10] J. Fisher, N. Piterman, A. Hajnal, and T.A. Henzinger. Predictive modeling of signaling crosstalk during *C. elegans* vulval development. *PLoS Computational Biology*, 3(5):e92, May 2007.
- [11] J. Fisher, N. Piterman, E.J.A. Hubbard, M.J. Stern, and D. Harel. Computational insights into *C. elegans* vulval development. *Proceedings of the National Academy of Sciences*, 102(6):1951–1956, February 2005.
- [12] S. J. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:11–121, 1980.
- [13] J. Koehler, G. Tirenni, and S. Kumaran. From business process model to consistent implementation: A case for formal verification methods. In *EDOC*, pages 96–106. IEEE Computer Society, 2002.
- [14] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.
- [15] M. Samer and H. Veith. Validity of CTL queries revisited. In *Proc. 17th CSL*, LNCS 2803, pages 470–483. Springer, 2003.
- [16] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
- [17] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

# Two-Dimensional Regular Expressions for Compositional Bus Protocols

Kathi Fisler

WPI Department of Computer Science  
Worcester, MA 01609 USA  
kfisler@cs.wpi.edu

**Abstract**—Bus and interconnect protocols contain a few core operations (such as read and write transfers) whose behaviors interleave to form complex executions. Specifications of the core operations should be flexible enough that simple composition operators suffice for capturing most interleavings, even in the presence of common hardware issues such as glitches. Oliveira and Hu proposed a form of pipelined regular expressions to specify atomic protocol compositions, but they abstracted away clocking and glitches. This paper uses the AMBA-2 specification to argue that a loosely-synchronized form of regular expressions handles such timing subtleties while retaining the simplicity of Oliveira and Hu’s pipelined compositions.

## I. INTRODUCTION

Last year, we attempted to formalize portions of the AMBA-2 interface specification for on-chip data transfers [1]. AMBA-2 uses timing diagrams extensively to present its protocols. Having worked on formalizing timing diagrams for many years [2], our goal was to explore whether formal diagrams were suitable for capturing such a complex specification. The experience was highly instructive: while formalized timing diagrams (ours or Amla et al’s [3]) could capture the individual diagrams in the specification, they were too concrete to capture the abstract compositional protocols underlying the diagrams. Oliveira and Hu’s pipelined regular expressions [4] avoid some of the diagrams’ shortcomings, but overconstrain timing issues that the diagrams support naturally. This paper describes our early-stage efforts to identify specification language constructs that capture the structure of AMBA-2 while drawing on the best features of both timing diagrams and pipelined regular expressions. As (readable) diagrams are generally more restrictive semantically than text, this exploratory work extends textual regular expressions. Finding a readable diagrammatic notation that supports the constructs we identify is an interesting problem for future work.

## II. THE STRUCTURE OF BUS SPECIFICATIONS

Figure 1 reproduces several AMBA-2 diagrams. The first row (document figures 5-9 and 5-11) shows basic read and write transfers, the second (document figure 5-12) shows a burst sequence of write transfers, and the third (document figure 5-13) shows an interleaved sequence of read and write transfers. The latter two diagrams are pipelined compositions of the first two. A good formalization of this specification would explicitly capture the basic transfers and exploit composition operators to derive the complex behaviors. A closer look at the

diagrams reveals several issues that a formal rendering must address to achieve this goal:

- 1) Pipelining is required, as seen in the overlapping instances of the write protocol in diagram 5-12.
- 2) The number of clock cycles between events must be allowed to vary between the basic and complex diagrams; clock specifications in the basic diagram may indicate alignment with edges rather than cycle counts.
- 3) The values on signals may differ when a basic protocol appears in composition. The basic write diagram (5-11) ends with PSEL low, but the burst diagram (5-12) preserves the prior high value across adjacent transfers.
- 4) Names equate values across signals in the diagrams.
- 5) Glitches and regions of potential signal instability (shown shaded) must be captured and can vary in length.

Other diagrams (not shown here) refine the basic protocols with error handling and other features. The observations in this paper are necessary, but not sufficient, for those cases.

Regular expressions are common for modeling protocols. As Oliveira and Hu argue [4], however, defining an interleaved sequence of protocols as a standard regular expression is problematic because such sequences capture parallel executions which vastly complicate the expressions. Furthermore, such expressions fail to reuse the specifications of the basic protocols. Their elegant proposal was to introduce a variant of the concatenation operator (denoted @ instead of ;) to indicate the point in a regular expression at which a parallel execution could begin. For example, the expression  $a;b@c$  allows a new instance of the expression (starting from  $a$ ) to begin concurrently with  $c$  (as the first term after the @ operator). With this notation, they can cleanly specify pipelined compositions of protocols whose steps execute in fixed numbers of cycles.

When the number of cycles between protocol steps is not fixed, however, the regular expressions show a key limitation. Consider the HADDR, HWDATA, and HREADY signals from the basic write protocol (5-11). Associating each character in an expression with a clock cycle, we could capture these as

$$\begin{aligned} \text{HADDR} &= \text{Addr}_1 ; dc^* \\ \text{HWDATA} &= dc ; \text{Data}_1 ; dc^* \\ \text{HREADY} &= H ; H ; dc^* \end{aligned}$$

(where  $dc$  stands for “don’t care” and  $H$  for “high”). These specifications, however, are too rigid to capture the instances of write transfers in the burst sequence (diagram 5-12). The

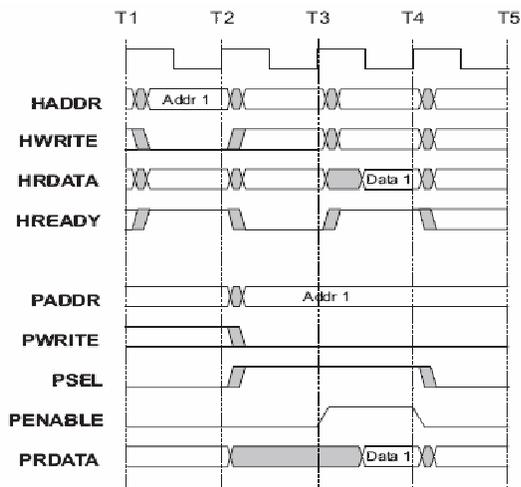


Figure 5-9 Read transfer to AHB

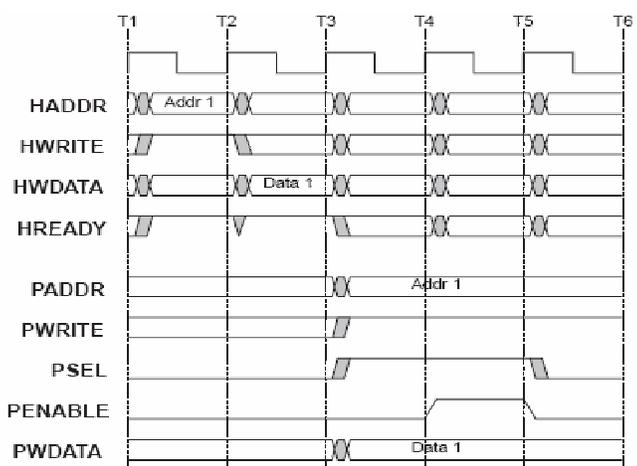


Figure 5-11 Write transfer from AHB

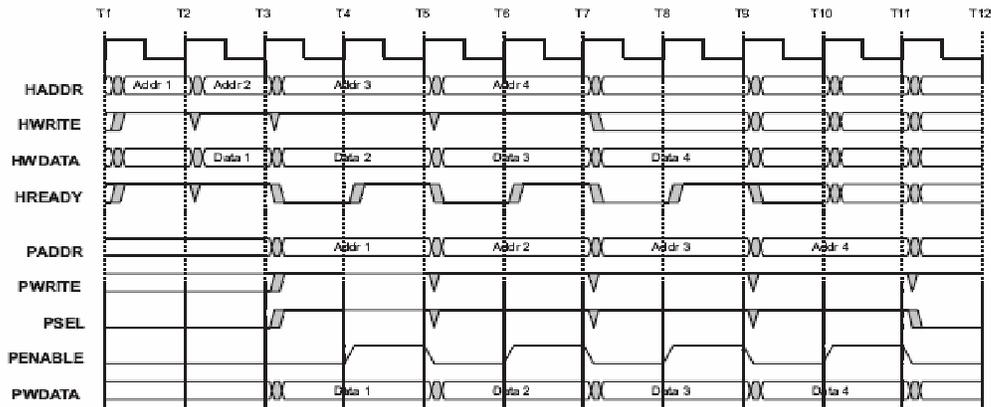


Figure 5-12 Burst of write transfers

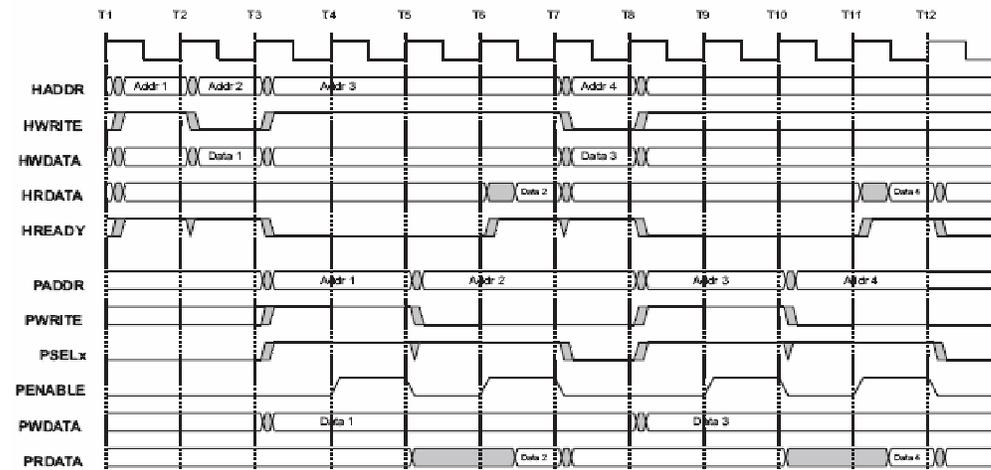


Figure 5-13 Back to back transfers

Fig. 1. Diagrams from the AMBA-2 specification [1] (pages 5-14 – 5-18) showing single read and write transfers (top), burst sequence of writes (middle), and sequence of read and write transfers (bottom). The lower two diagrams consist of pipelined instances (compositions) of the behaviors in the two diagrams in the top row. The figure numbers listed under the diagrams are from the original specification; we refer to the original numbers in the rest of the paper.

regular expressions require the *Data* value on HWDATA to occur in the next clock cycle after the *Addr* value on HADDR, but that does not happen for *Addr*<sub>3</sub>/*Data*<sub>3</sub>. Each of *Addr*<sub>3</sub> and *Data*<sub>3</sub> may also extend beyond the single clock cycle shown in the basic diagram. This suggests not equating clock cycles with steps in the regular expression. In similar vein, the regular expression for HREADY contains two cycles at high value, with the first synchronized with *Addr*<sub>1</sub>. In the burst diagram, the HREADY high value for *Addr*<sub>2</sub> occurs simultaneously with *Addr*<sub>3</sub> on HADDR (due to pipelining), and only lasts a single clock cycle. Rewriting the regular expression for HREADY to expect only a single high value would address the latter problem, but not the synchronization problem.

As these examples illustrate, the synchronization implicit in regular expressions running concurrently is fundamentally limiting for protocols in which relative position of terms matters more than precise cycle counts. Regular expressions naturally capture individual signals, but their composition needs to be more nuanced. Modeling potential glitches exacerbates the limitations of regular expressions. A combination of pipelined regular expressions with the more flexible alignment of events natural in timing diagrams seems promising. Specifically, we decouple concatenation from clocking using synchronization and ordering constraints inspired by timing diagrams to relate atomic expressions across regular expressions. The resulting combination of inter-signal and cross-signal specification, exemplified by timing diagrams, inspires our term “two dimensional regular expressions”.

### III. 2D REGULAR EXPRESSIONS

We develop our 2D regular expression from the ground up, working with the diagrams in figure 1. The basic read protocol (diagram 5-9) suggests the atomic terms needed to capture signals: high and low bit values (seen on HREADY), denoted *H* and *L*, respectively; named bus values (such as *Addr*<sub>1</sub> on HADDR), denoted by variable names; unnamed but stable values (on HADDR after *T*<sub>2</sub> and HREADY after *T*<sub>4</sub>), denoted by *sd* (for “stable but don’t care”) and potentially unstable values (the shaded areas on all signals), denoted by *uv* (for “unstable value”). We use the standard operators of ; for concatenation, + for choice, and \* for iteration, plus Oliveira and Hu’s @ for pipelined concatenation. The diagrams we have studied so far rarely, if ever, nest uses of the \* operator, so the expressions for signals tend to be relatively simple.

Using these atomic terms, the HADDR signal in the read diagram appears to correspond to

$$sd; uv; Addr_1 @ uv; sd; uv; sd; uv; sd.$$

This expression expects three (potentially distinct) stable bus values to occur on HADDR after *Addr*<sub>1</sub> appears. Furthermore, the corresponding *Data*<sub>1</sub> value on HRDATA is expected after two of those stable values have appeared. In the read/write transfer sequence (diagram 5-13), the second transfer (starting with *Addr*<sub>2</sub>) is a read transfer and should be consistent with this expression. There, *Data*<sub>2</sub> occurs after only one *uv* value on HADDR. The proposed HADDR expression is therefore

overly specific relative to how the transfer gets instantiated in sequence. The essence of the problem is that the basic read protocol must specify the *potential* for unstable values, whereas the semantics of regular expressions *requires* the unstable period to occur. To retain flexibility for composition, then, *the regular expressions for a signal should capture only required behavior*, leaving optional behavior to auxiliary annotations. We accordingly eliminate the *uv* values from the regular expression. For flexibility, we also exploit \* to avoid fixing the number of stable regions that will follow *Addr*<sub>1</sub>. The resulting specification of HADDR is *sd; Addr\_1 @ sd\**. Figure 2 shows the full expression for the basic read diagram, and is explained in the rest of this section.

While our treatment of HADDR suggests that *uv* values should never appear in the regular expressions for signals, PRDATA shows an example where *uv* is appropriate. On PRDATA, the long period of instability is a standard part of the protocol, and should be included in the regular expression. We capture PRDATA as *sd; uv; Data\_1; sd*. The decision of whether or not to model a shaded region as *uv* depends on whether the depicted period of instability will necessarily occur.

The protocols require certain atomic terms to happen either concurrently or in particular orders. In the core read protocol (diagram 5-9), *Data*<sub>1</sub> appears on HRDATA after the rising transition on HREADY, and the fall of HREADY aligns with the end of when *Data*<sub>1</sub> is stable. The following two annotations capture this, where *S*<sub>*T*<sub>*i*</sub></sub> denotes the *i*<sup>th</sup> concatenation operator (; or @) in signal *S* (counting from 1):

$$\begin{aligned} & \text{Order}(\text{HREADY}_{T3}, \text{HRDATA}_{T1}) \\ & \text{Event}(\text{HREADY}_{T4}, \text{HRDATA}_{T2}) \end{aligned}$$

These have the obvious semantics: *Event* requires its arguments to be concurrent and *Order* requires the second term to occur strictly after the first. Optional numeric arguments to *Order* give lower and upper bounds on the time between the events. Additional *Event* specifications align portions of HADDR, HWRITE, and HREADY as shown in figure 2.

This leaves handling the potentially unstable regions (glitches) in the signals. The unstable regions omitted from HADDR occur around transitions. The regular expressions defining the signals suggest that transitions consume no measurable time: expression *H; Addr\_1* has *Addr*<sub>1</sub> stable in the time unit immediately following *H*. To support glitches, we must relax this. Our transitions have distinct start and end times that may be separated by more than one unit of time. Often, the fact that transitions may take time is more important than the amount of time that they take. Annotation *GlitchTrans*(*S*<sub>*T*<sub>*i*</sub></sub>) indicates that the *i*<sup>th</sup> transition on signal *S* may take multiple units of time. Unlike the *Event* and *Order* annotations, *GlitchTrans* annotations may be ignored (desirable for analyses that abstract away timing): each *GlitchTrans* extends the set of traces that satisfy the expression, whereas the other annotations restrict the set of satisfying traces.

The read diagram contains two additional pieces of information about glitches: many glitch regions span the same period of time, and that period lies within the high clock period (the unnamed signal at the top of the diagram). *Event* annotations

```

HADDR = sd; Addr1 @ sd* [* allows multiple transfers]
HWRITE = sd; L @ sd*
HRDATA = sd* ; Data1 @ sd
HREADY = sd; H; L; H @ sd

Event(HADDRT1, HWRITET1, HREADYT1)
Event(HADDRT2, HWRITET2, HREADYT2)
Event(HRDATAT2, HREADYT4)
Event(HRDATAT1-first, HREADYT3-first)
Order(HRDATAT1, HREADYT3-last)

ClockSignal(CLK)
GlitchTrans(HREADYT1)
Constrain(HADDRT1-first, HADDRT1-last, CLK = H)
[similar constraints for remaining glitch regions]

```

Fig. 2. 2D regular expression for the basic read transfer protocol (AMBA diagram 5-9). The ClockSignal annotation indicates that the named signal oscillates between high and low values as a clock is expected to do.

align the endpoints of glitch regions ( $S_{Ti-first}$  and  $S_{Ti-last}$  refer to the start and end times of the  $i^{\text{th}}$  transition on signal  $S$ ). Containing these regions within positive clock periods is harder with the annotations given so far. Order constraints require naming the clock transitions, which is difficult since the number of clock cycles used in a composed transaction may vary. Instead, we introduce an annotation  $\text{Constrain}(e_1, e_2, s = v)$  that requires signal  $s$  to hold value  $v$  between events  $e_1$  and  $e_2$ . An example appears in figure 2. An Order annotation on the start and end times could bound glitch durations.

This discussion of glitches, events, and order constraints potentially masks the more substantive points about using regular expressions to effectively model the AMBA-2 protocols compositionally. The protocols as shown in the diagrams capture different information at different granularities of clocking: the glitch regions fall between the clock cycles governing the logical heart of the protocols. Multiple clocking granularities are also useful for capturing the essence of high-level signal behaviors: a value that spans only one clock cycle in the basic protocol diagram may span several in a composed sequence, while corresponding to a single logical concept in the protocol. Other AMBA protocols (such as multi-address burst commands from a bus master [AMBA-2 figure 3-6, not shown]) contain signals whose values change upon events in other signals. Decoupling clocking from the notion of time implicit in the regular expressions seems essential to handling these problems cleanly; some system of synchronization and ordering constraints is needed to make up the difference. Various notations suffice, from explicit Event and Order constraints on all related pairs to statements that assign signals to clocks and rely on the usual concurrent semantics of regular expressions. The differences are syntactic and irrelevant for this paper.

#### IV. PERSPECTIVE AND DISCUSSION

This paper describes the early stages of a project to formalize interface specifications while capturing the relationships

between their constituent protocols. AMBA-2 develops its protocols iteratively, refining basic behaviors through successive sections of the document. Using the framework outlined here, AMBA-2 figures 5-12 and 5-13 are instances of the protocols shown in figures 5-9 and 5-11; only the latter two (the basic protocols) must be specified explicitly. We are trying to identify composition constructs that enable this sort of reuse across the entire specification without sacrificing readability or ease of abstraction for different analysis tasks. Two principles have emerged from our work so far: basic protocol specifications must treat clocking and signal values sufficiently abstractly to support pipelined composition, and clocking overall must be decoupled from concatenation to enable specification at different levels of time abstraction.

Pipelined regular expressions make it feasible to use regular expression-like notation for some protocol compositions, but lack the fine-grained timing control needed to compositionally specify AMBA-2. Timing diagrams handle multiple time granularities well, but can be a bit too concrete in the face of arbitrary \* operators and cross-signal constraints that depend on sequences of events. We could add a pipelining notation to timing diagrams to handle the examples in this paper, but that would be premature. Other AMBA-2 protocols that extend the core operators (such as those for error handling) demand different composition operators. We want to identify these operators before choosing a notation. Aspect-oriented programming [5] offers composition operators for fine-grained behavior extensions that are promising for the additional protocols. Balancing the rich information structure of the diagrams while retaining sufficient abstraction and readability is an important area for future work.

We are also applying our constructs to AXI, HyperTransport, and the OVL assertion language (also specified largely diagrammatically). The latter effort is promising, but the former two lack the explicit iterative development style of AMBA-2. We are presenting this work as a short paper in hopes of gaining early insight into what determines the high-level structure of an interface protocol specification, and what kinds of high-level structures lead to designer-friendly specifications.

*Acknowledgements:* Mike Gordon proposed studying the structure of timing diagrams in bus specifications via AMBA-2. NSF grants CCR-0132659 and CCR-0305834 funded this work. The reviewers provided several thought-provoking comments.

#### REFERENCES

- [1] ARM Limited, "AMBA specification (rev 2.0)," May 1999, <http://www.arm.com/products/solutions/amba2overview.html>.
- [2] K. Fisler, "Timing diagrams: Formalization and algorithmic verification," *Journal of Logic, Language, and Information*, vol. 8, pp. 323–361, 1999.
- [3] N. Amla, E. A. Emerson, and K. S. Namjoshi, "Efficient decompositional model checking for regular timing diagrams," in *IFIP Conference on Correct Hardware Design and Verification Methods*, 1999.
- [4] M. T. Oliveira and A. J. Hu, "High-level specification and automatic generation of IP interface monitors," in *International Conference on Design Automation*. ACM Press, 2002, pp. 129–134.
- [5] "Aspect oriented programming (article series)," *Communications of the ACM*, vol. 44, no. 10, Oct. 2001.

# A Quantitative Completeness Analysis for Property-Sets

Martin Oberkönig  
Computer Systems Lab  
Darmstadt University of Technology  
Darmstadt, Germany  
oberkoenig@rs.tu-darmstadt.de

Martin Schickel  
Computer Systems Lab  
Darmstadt University of Technology  
Darmstadt, Germany  
schickel@rs.tu-darmstadt.de

Hans Eveking  
Computer Systems Lab  
Darmstadt University of Technology  
Darmstadt, Germany  
eveking@rs.tu-darmstadt.de

**Abstract**—This paper defines a quantitative metric of the completeness of a formal specification. A “good” (formal) specification is already needed in the beginning of the development process to prevent cost-intensive corrections of errors found in the late phases of a design process. A quantitative analysis method is presented to evaluate whether a specification is “good”. The method considers only the formal specification, an implementation (design) is not necessary. Thus, an analysis of a specification can be directly done. The properties of the specification are transformed into a normal form to calculate the metric. Experimental results of the analysis method are given, e.g., metrics for specifications of the AMBA AHB bus.

## I. INTRODUCTION

As digital circuits become larger, more complex and safety critical, the demand for error-free designs has increased dramatically. A “good” specification is already needed in the beginning of the development process in order to avoid the most cost-intensive corrections of errors found in the late phases of a design process. In property-based design, properties written in a formal property language like PSL [1] or some temporal logic are used as a specification. Properties can be employed even in the very early design phases to generate hardware-prototypes (*Cando-Objects* [2], [3]). To evaluate whether a specification is “good” or not, an analysis method is needed.

On the one hand, one wants to know if there are enough properties to specify the intended behavior completely. For this purpose, it is useful to know to which degree the outputs of a system are specified. A specification is typically incomplete in the early design phases, so making a difference between “an output is completely specified” and “an output is never mentioned” is important. If, on the other hand, a design is not available the properties may be inconsistent, i.e., contradicting one another. During the later verification process only one of the contradicting properties can be satisfied which might not be obvious in the early specification process.

In the field of simulation, it has been state of the art for more than ten years to employ several kinds of coverage analyses. These consider, for example, *code coverage* [4] or *transition coverage* [5] to evaluate the exhaustiveness of test suites. However, these techniques cannot be directly applied to the domain of formal verification.

This paper defines a quantitative metric of the completeness of a specification used for formal verification. The presented

method considers only the specification, an implementation is not needed.

## II. RELATED WORK

Among the first to discuss completeness of the formal verification process were Katz et al. [6]. They define completeness of a specification with respect to a given implementation.

Hoskote et al. propose an approach for a metric in model checking [7]. The algorithm measures coverage with respect to an observed signal in an implementation. The transfer of the traditional metrics from simulation-based verification to formal verification with respect to a given design is described in [8].

Große et al. propose an approach to find specification holes in a property-set [9]. The method is useful and efficient in some situations, but has some serious drawbacks: It works only with a design and only one gap can be shown at a time, making it impossible to obtain an overview of the specification status.

In [10] K. Claessen introduces a method to analyze the coverage of safety property lists. An observer is generated from a list of properties. The outputs of two identical observers are compared during a model checking process. The method fulfills one of our goals, because no design is needed for the analysis, but only a single gap can be identified at each step.

The coverage of RTL properties against higher level architectural properties or high-level fault models is discussed in [11] and [12]. These methods cannot tell anything about one property-set itself or about an overall verification status in form of a metric.

As a summary, there are already several metrics for formal verification, but all need a design to work on. A quantitative measure of specification completeness regardless of a design is still needed.

## III. COVERAGE, COMPLETENESS AND CONSISTENCY

In simulation-based verification several coverage metrics exist (see e.g. [4], [5], [8]). However, the term *coverage* cannot be simply adopted for formal verification. In formal verification a specification is always verified for the complete model. Thus, the question is not ‘what part of the design

is covered',but 'is all relevant functionality included in the specification'. Hence, we use the term *completeness* instead.

We propose a completeness analysis that calculates a *degree of determination* of the signals involved in the specification. Since many specification languages are property-based, we will assume that a specification is composed of properties.

A signal is fully determined, if for each input and internal state combination one specific value is assigned to the signal by the set of properties.

Properties can generally be divided into two groups: One of them restricts a signal bit to '0' in a certain situation, another restricts it to '1'. These two groups are very similar to the drivers of a CMOS circuit: The PMOS net  $v_1$  forces the output to a '1' if active while the NMOS net  $v_0$  will connect the output to the '0'. This analogy motivates the further procedure. In a well-defined CMOS circuit

$$v_0 = \neg v_1 \quad (1)$$

is always valid.

Generally, the determination-function and the consistency-function of a CMOS circuit are given by:

$$determination = v_0 \vee v_1 \quad (2)$$

$$consistency = v_0 \wedge v_1 \quad (3)$$

If the determination-function is equal to '1' then the output is fully determined (complete). In any other case it is partially determined or not determined at all. The complement of the determination-function characterizes all situations in which the output is *undetermined*. The consistency-function must always be '0', otherwise the circuit is inconsistent. All situations for which the drivers are inconsistent are identified by the consistency-function.

The determination-function and the consistency-function can be directly adopted for the analysis of formal property-sets if the properties are in the following form:

$$P_0 : \quad v_0 \implies \neg signal \quad (4)$$

$$P_1 : \quad v_1 \implies signal \quad (5)$$

#### IV. DEFINITION OF THE COMPLETENESS METRIC

A measurement of the degree of signal determination can be derived from the zero-function  $v_0$  and the one-function  $v_1$  of every bit of every output and signal.

**Definition:** If the determination-function of a signal is '1' then the signal is completely determined. In any other case, the following metric gives the percentage of determined situations:

$$degree\ of\ determination = \frac{\#minterms}{2^n} \quad (6)$$

where  $n$  is the number of variables in the support and  $\#minterms$  is the number of satisfying assignments to all variables of the determination-function, respectively.

If the consistency-function is not '0', then inconsistencies have been found. A metric for a consistency degree does not make sense, because even a single situation which leads to the assignment of contradictory values to one signal must be corrected.

#### V. NORMALIZATION

Obviously, most specifications do not write properties in a form as described above. We therefore have developed a method to transform many arbitrarily written properties into a normal form. We assume that the set of properties consists of safety properties of the form  $AG(P)$ , where  $P$  is of the form

$$P : A(I_{tmin}, \dots, I_{tmax}, S_{tmin}, \dots, S_{tmax}, O_{tmin}, \dots, O_{tmax}) \\ \implies C(I_{tmin}, \dots, I_{tmax}, S_{tmin}, \dots, S_{tmax}, O_{tmin}, \dots, O_{tmax}) \quad (7)$$

The assumption  $A$  and the commitment  $C$  may contain input signals ( $I$ ), internal state signals ( $S$ ) and output signals ( $O$ ) at different timepoints of a finite time-window  $tmin\dots tmax$ .  $tmin$  is the earliest timepoint and  $tmax$  the latest timepoint used in the property.

The properties are transformed into an appropriate representation by means of a multi-step normalization process adopted from [2] for the creation of so-called Cando-Objects.

In a first step, all signals with references to timepoints less than  $tmax$  are transferred to the left hand side of the implication. Afterwards only signals at timepoint  $tmax$  are on the right side (Eq. (8)). This is the intuitive form of a causal specification: the past and the present imply the future.

$$P : A(I_{tmin}, \dots, I_{tmax-1}, S_{tmin}, \dots, S_{tmax-1}, \\ O_{tmin}, \dots, O_{tmax-1}) \implies C(I_{tmax}, S_{tmax}, O_{tmax}) \quad (8)$$

The properties are now in the same normal form as described in [2]. The commitment of a property may consist of an arbitrary boolean expression involving input, internal and output signals. For the calculation of the  $v_0$ 's and  $v_1$ 's it is furthermore necessary to have an equivalent set of properties with exactly one output (or internal state) signal in each commitment. The commitment of a property is first transformed into a conjunctive normal form (CNF). Then the property is split up into a number of properties with the same original assumption using each clause as a separate commitment. For instance,

$$A \implies Clause_1 \wedge Clause_2$$

will be transformed into

$$A \implies Clause_1 \quad \text{and} \quad A \implies Clause_2.$$

At this point of the normalization process, the commitments of all properties consist of a single clause only. This clause in turn is always a disjunction of literals. In a last step, the disjunctions are removed as shown in the following example:

$$A \implies S_1 \vee S_2$$

will be transformed to

$$A \wedge \neg S_2 \implies S_1 \quad \text{and} \quad A \wedge \neg S_1 \implies S_2.$$

This transformation might generate properties which have an input signal as a commitment. Such properties are eliminated since it is not possible to force an input signal to a value.

Now the commitments of all properties refer to only one single internal or a single output signal at the latest timepoint  $tmax$ .

$$P : A(I_{tmin}, \dots, I_{tmax}, S_{tmin}, \dots, S_{tmax}, O_{tmin}, \dots, O_{tmax}) \implies C(S_{tmax}/O_{tmax}) \quad (9)$$

This normalized property-set will be used for the calculation of the determination-function needed to obtain the metric and for the calculation of the consistency-function.

## VI. CALCULATION OF THE ZERO- AND ONE-FUNCTIONS

The normalized set of properties contains both properties with a positive commitment, which forces a signal to '1', and properties with a negative commitment, forcing a signal to '0'. To build a signal's zero-function  $v_0$ , all assumptions forcing the signal to '0' are disjoint. Likewise, the one-function  $v_1$  of a signal is the disjunction of the properties that force the signal to '1'.

For example, the properties

$$\begin{aligned} A_1 &\implies \neg S_1 & \text{and} & & A_2 &\implies \neg S_1 \\ A_3 &\implies S_1 & \text{and} & & A_4 &\implies S_1 \end{aligned}$$

result in  $v_0 = A_1 \vee A_2$  and  $v_1 = A_3 \vee A_4$  for the signal  $S_1$ .

The assumptions now describe all situations, in which a signal is forced to a certain value. Thus, the disjunction  $v_0 \vee v_1$  (determination-function) represents all situations in which a signal has a defined value. In these cases the signal is determined.

## VII. PROPERTY DEPENDENCIES

As shown in the section before, the  $v_0$ 's and  $v_1$ 's are calculated by building the sum of all assumptions. The normalization process might generate properties with assumptions referencing internal or output signals at  $tmax$  (see Eq. (9)). These signals are currently analyzed for determination. An assumption which references such a signal cannot be simply added to  $v_0$  or  $v_1$  because the status of determination of this assumption is unclear. The following example is used to illustrate the problem:

$$P : true \implies a \equiv b$$

The outputs  $a$  and  $b$  obviously do not have a defined value. The normalization of this property leads to the four properties

$$\begin{aligned} P_1 : & a \implies b & \text{and} & & P_2 : & \neg a \implies \neg b \\ P_3 : & b \implies a & \text{and} & & P_4 : & \neg b \implies \neg a. \end{aligned}$$

If the fact, that the outputs  $a$  and  $b$  are not determined, was neglected during the calculation of the  $v_0$ 's and  $v_1$ 's the result would be 100% determination of both  $a$  and  $b$ . This would contradict both signal's degree of freedom as specified by the original property.

In such a case, an output can only be used in the assumption of a property if that output is at least partially determined by additional properties.

A property dependency graph as shown in Fig. 1 is used to solve the problem. The graph shows how the outputs (or

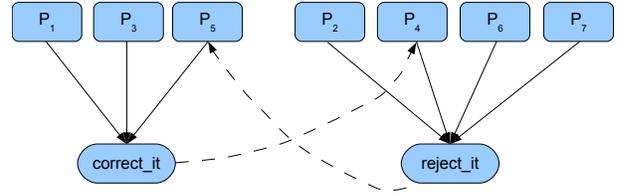


Fig. 1. Dependency Graph of the AEC

TABLE I  
AEC PROPERTIES (ORIGINAL)

P1:	$\neg error_{t+1} \implies \neg correct\_it_{t+1} \wedge \neg reject\_it_{t+1}$
P2:	$\neg error_t \wedge error_{t+1} \wedge \neg multiple_{t+1} \implies correct\_it_{t+1}$
P3:	$true \implies \neg correct\_it_{t+1} \vee \neg reject\_it_{t+1}$
P4:	$error_t \wedge error_{t+1} \wedge \neg multiple_{t+1} \implies reject\_it_{t+1}$
P5:	$error_{t+1} \wedge multiple_{t+1} \implies reject\_it_{t+1}$

internal signals) depend on the properties. A directed edge exists from property  $P_i$  to node  $x$  iff  $x$  occurs as a literal in the commitment of  $P_i$ . Vice versa, a directed edge exists from node  $x$  to property  $P_i$  iff  $x$  is in the support of the assumption part of  $P_i$ .

As a consequence of such dependencies the calculation of the  $v_0$ 's and  $v_1$ 's is organized as an iterative process. At first all determined assumptions (with only inputs and signals from  $tmin \dots t$ ) are added to the corresponding  $v_0$ 's and  $v_1$ 's. This may partially determine some outputs which are used to further determine the assumptions of other outputs. If all completeness information has been propagated, the  $v_0$ 's and  $v_1$ 's do not change any more. Then a fixed point is reached and the algorithm is completed. The metric for each output and internal signal is calculated according to Eq. (6).

## VIII. EXPERIMENTAL RESULTS

We analyzed several property-sets to validate the described method. The first component was an ATM error controller (AEC) already used in [13] for coverage analyses. The AEC is a small circuit that can easily be formally specified. The system is small enough so that our property-set completeness analysis can be presented in full detail. Later some more results of larger, more significant property-sets will be given.

The AEC has two inputs ( $error$  and  $multiple$ ) which tell the controller whether an ATM cell has an error and whether multiple errors have occurred. The outputs ( $reject\_it$  and  $correct\_it$ ) signal if an error can and should be corrected or if the ATM cell should be rejected. This system is formally described by five properties (see Tab. I). The properties shown in Tab. II are the same properties but after the normalization procedure described in Section V. The conjunctive ( $p1$ ) and disjunctive ( $p3$ ) right hand sides of the implications are now split. Every right hand side of the normalized properties references only a single signal which in this case is always an output.

Fig. 1 shows the corresponding dependency graph of the normalized AEC properties and the two outputs. The properties  $p4$  and  $p5$  depend not only on inputs and signals at past timepoints but also on outputs at the latest timepoint

TABLE II  
AEC PROPERTIES (NORMALIZED)

P1:	$\neg error_{t+1} \implies \neg correct\_it_{t+1}$
P2:	$\neg error_{t+1} \implies \neg reject\_it_{t+1}$
P3:	$\neg error_t \wedge error_{t+1} \wedge \neg multiple_{t+1} \implies correct\_it_{t+1}$
P4:	$correct\_it_{t+1} \implies \neg reject\_it_{t+1}$
P5:	$reject\_it_{t+1} \implies \neg correct\_it_{t+1}$
P6:	$error_t \wedge error_{t+1} \wedge \neg multiple_{t+1} \implies reject\_it_{t+1}$
P7:	$error_{t+1} \wedge multiple_{t+1} \implies reject\_it_{t+1}$

TABLE III  
EXPERIMENTAL RESULTS (OVERVIEW)

Component	Prop.	normal. Prop.	Analysis Time
ATM Error Controller	5	7	0.1 s
AMBA Slave	8	497	0.69 s
AMBA Master	20	3290	7.3 s

$t + 1 = tmax$ . The two properties cannot contribute to the  $v_0$ 's and  $v_1$ 's of the two outputs in the first iteration. After their (partial) determinations have been calculated, the  $v_0$ 's and  $v_1$ 's of the outputs can be used in later iterations.

When all  $v_0$ 's and  $v_1$ 's are built, the determination-function  $v_0 \vee v_1$  is calculated to obtain the degree of determination of each output. The consistency-function  $v_0 \wedge v_1$  may show potential inconsistencies. For the AEC no inconsistencies are found and the two outputs are completely determined.

As more advanced systems we analyzed specifications of the master and the slave components of the AMBA AHB protocol [14]. Both components were not completely specified. Tab. III gives the analysis time and the number of original and normalized properties. The degree of determination of each output and internal signal is given in Tab. IV. As the two AMBA AHB property-sets are only protocol specifications and do not contain any information about the actual implementation and environment, some signals have a very low degree of determination. For instance, the data outputs of the two components get their values from the environment, and are not defined by the protocol compliance. In addition, some features like protected control via *hprot* are not supported.

## IX. CONCLUSIONS

In this paper we have introduced a completeness analysis method for formal property-sets. The calculated metric quantitatively describes the degree of determination of the output signals and internal signals mentioned in a specification. The completeness metric is derived only from the properties itself, and gives an overall view of the status of a specification. The property normalization process presented allows to analyze arbitrarily written properties.

If the analysis approves the completeness of the formal specification then this does not mean that the specification is perfect. It only proves, that all outputs of the design are determined, i.e., having a defined value in every possible situation. If the formal properties do not adhere to an original informal specification, then even the full determination does not lead to an error-free specification. This must be considered when interpreting the completeness metric.

TABLE IV  
EXPERIMENTAL RESULTS (DETAILED)

Component	Signal	Output / Internal Signal	Determination
AEC	reject_it	out	100%
	correct_it	out	100%
AMBA Slave	act_master(3..0)	int	100%
	split_master(15..0)	int	49%
	selected	int	100%
	hsplit(15..0)	out	49%
	hresp(1..0)	out	21%
	hready	out	21%
	hrdata(31..0)	out	0%
AMBA Master	cnt_s(3..0)	int	7%
	htrans(0)	out	66%
	htrans(1)	out	54%
	hburst(2..0)	out	1%
	haddr(31..0)	out	1%
	hwrite	out	1%
	hsize(2..0)	out	1%
	hprot(3..0)	out	1%
	hlock	out	0%
	hbusreq	out	0%
	hwdata(31..0)	out	3%

## REFERENCES

- [1] Accellera, *Property Specification Language, Reference Manual, Version 1.1*. Accellera, 2004.
- [2] M. Schickel, V. Nimbler, M. Braun, and H. Eweking, "On consistency and completeness of property-sets: Exploiting the property-based design-process," in *Proc. of FDL*, 2006.
- [3] H. Eweking, M. Braun, M. Schickel, M. Schweikert, and V. Nimbler, "Multi-level assertion-based design," in *Proc. of Memocode'07*, 2007.
- [4] K. T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *DAC '93: Proceedings of the 30th international conference on Design automation*. New York, NY, USA: ACM Press, 1993, pp. 86–91.
- [5] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture validation for processors," in *ISCA*, 1995, pp. 404–413.
- [6] S. Katz, O. Grumberg, and D. Geist, "'have i written enough properties?' — a method of comparison between specification and implementation," in *CHARME*, 1999, pp. 280–297.
- [7] Y. V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *DAC*, 1999, pp. 300–305.
- [8] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for formal verification," in *Correct Hardware Design and Verification Methods (CHARME)*, 2003, pp. 111–125.
- [9] D. Große, U. Kühne, and R. Drechsler, "Estimating functional coverage in bounded model checking," in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2007.
- [10] K. Claessen, "A coverage analysis for safety property lists," Presentation at Workshop on Designing Correct Circuits (DCC), 2006.
- [11] P. Basu, S. Das, A. Banerjee, P. Dasgupta, P. Chakrabarti, C. R. Mohan, L. Fix, and R. Armoni, "Design-intent coverage - a new paradigm for formal property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2006.
- [12] S. Das, A. Banerjee, P. Basu, P. Dasgupta, P. Chakrabarti, C. R. Mohan, and L. Fix, "Formal methods for analyzing the completeness of an assertion suite against a high-level fault model," in *Proceedings of the 18th International Conference on VLSI Design*, 2005.
- [13] P. Molitor and S. Ruppe, "A coverage measure for bounded model checking," Martin-Luther Universität Halle, Tech. Rep., 2003.
- [14] *AMBA Specification (Rev 2.0)*. ARM Limited, 05/1999.

# Automated Extraction of Inductive Invariants to Aid Model Checking

Michael L. Case   Alan Mishchenko   Robert K. Brayton  
 Department of EECS, University of California, Berkeley  
 {casem, alanmi, brayton}@eecs.berkeley.edu

**Abstract**—Model checking can be aided by inductive invariants, small local properties that can be proved by simple induction. We present a way to automatically extract inductive invariants from a design and then prove them. The set of candidate invariants is broad, expensive to prove, and many invariants can be shown to not be helpful to model checking. In this work, we develop a new method for systematically exploring the space of candidate inductive invariants, which allows us to find and prove invariants that are few in number and immediately help the problem at hand. This method is applied to interpolation where invariants are used to refute an error trace and help discard spurious counterexamples.

## I. INTRODUCTION

Formal model checking of safety properties in a sequential machine is often intractable. In practice, prior knowledge about the design and/or property being checked greatly reduces verification time. Often, this knowledge is all that makes formal verification practical.

Frequently this prior knowledge comes in the form of hints from the designer, but it is difficult to identify helpful hints, and there is a real danger of either expressing hints that do not help or that are not true. Extraneous hints may slow down the verification tool by consuming valuable resources; all hints must be proved to verify that the designer did not err, possibly increasing the complexity of the overall problem.

An ideal solution would be to automatically extract useful hints from the design. Such a method should focus only on those hints that immediately help the verification and whose proof is simple. This paper proposes a method to do exactly that.

We focus on simple design properties called *inductive invariants*. These are properties that hold in every reachable state and can be proved by *simple induction*, temporal induction using only a 1-step (current state, next state) model. Often a verification tool can be assisted by the knowledge that a small set of states in the design is unreachable. For example, if an abstraction reaches a bad state, we would like to know if any state on the error trace is unreachable. If there exists an inductive invariant that demonstrates this unreachability, then the counterexample is spurious, and there is no need to refine the model.

To harness this idea, we have built a tool that is able to show that a single, user-specified state is unreachable. It does this by finding and proving inductive invariants. If for some reason it is unable to complete the proof of the invariants, it will find and prove other, secondary invariants that enable the

first proof to proceed. This method gives a hierarchy of proofs that when complete will yield a set of inductive invariants  $\{P\}$  with the following properties:

- $\bigwedge_{p \in \{P\}} p$  implies that the user-specified state is unreachable.
- $\bigwedge_{p \in \{P\}} p$  can be proved with simple induction

This paper provides the theory behind this invariant generation method and explores how specific inductive invariants can help interpolation, a method for unbounded verification of safety properties.

## II. RELATED WORK

An excellent background on formal verification can be found in [1], and [2] is a good overview of modern unbounded verification techniques. [3] describes the status of formal verification inside IBM. It describes the problem of extraneous hints and specifications in the following phrase: "person-months spent developing formal specs, merely to choke [the] FV tool." The above papers establish the basics and identify one of the key problems.

This paper will discuss a specific verification algorithm called interpolation. This method, originally proposed by McMillan in [4], is fast and compares favorably to other techniques [5], and thus it was chosen for study in this work.

This paper builds on our previous work [6] in which Boolean implications between design nodes are discovered through random simulation and proven with simple induction. The method was inspired by techniques proposed by van Eijk in [7]. Bjesse proposes a way to strengthen simple induction in [8]. We will strengthen simple induction in a different manner.

Finally, this work was motivated by Wedler [9] who discusses using implications between design nodes to form inductive invariants. He also attempts to strengthen simple induction in a manner similar to Bjesse. However, that work is limited in scope (1-hot machines), while our work is more general and provides a solid theoretical foundation.

## III. APPLICATIONS OF INDUCTIVE INVARIANTS

Inductive invariants are useful in many applications. Here we examine two such applications: simple induction and interpolation. By exploring these applications, we can expose some of their weaknesses and motivate the need for inductive invariants.

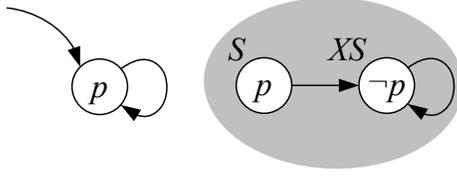


Fig. 1. The shaded states are unreachable but will make an inductive proof of  $p$  impossible.

### A. Simple Induction

Simple induction is a method of proving that a property holds for all reachable states. It is easy to formulate and often executes quickly. It is an incomplete method but can be strengthened by the introduction of inductive invariants.

First, we define some basic notation. If  $S$  is a state and  $p$  a property then  $S \models p$  shall mean that  $S$  satisfies the property  $p$ . No guarantees are made for other states. We also refer to states  $XS$  as any state that can be reached from  $S$  in one state transition. In general,  $XS$  is not unique.

In simple induction, a property  $p$  is proven in a two step process:

**Base Case:** Show  $\forall$  initial states  $i$ , that  $i \models p$ .

**Inductive Step:** Show  $\forall S$  s.t.  $S \models p$ ,  $\forall XS$ , that  $XS \models p$

This is typically implemented by unrolling the transition relation and then formulating a SAT problem to check both conditions of the proof.

The technique is incomplete since there are properties that hold in every reachable state which simple induction will fail to prove. For example, Figure 1 shows a state transition graph on which a proof of  $p$  will fail. The shaded states are unreachable, but because of these unreachable states  $\exists S$  and  $XS$  such that  $S \models p$  and  $XS \not\models p$ . The inductive step fails.

---

#### Algorithm 1 Modified simple induction.

---

```

1: // Let  $p$  be the property to be proved.
2: if  $\exists$  initial state  $i$ ,  $i \not\models p$  then return “falsified”
3: if  $\exists$  possibly reachable  $S, XS$  s.t.  $S \models p$ ,  $XS \not\models p$  then
4:   if can prove  $S$  or  $XS$  unreachable then // Section IV
5:     record new invariants, goto 3 // See Section IV-D
6:   return “inconclusive”
7: end if
8: return “verified”

```

---

To prove  $p$  in Figure 1, we may try  $k$ -step induction as described in [8], but in [6] we found this to be a very expensive solution. Instead, suppose we are able to find an inductive invariant that shows that either  $S$  or  $XS$  is unreachable. If known-unreachable states are disallowed from entering the inductive step (with an extra constraint on the SAT solver), then simple induction will be able to prove  $p$ . This is demonstrated in Algorithm 1 where the standard simple induction algorithm is improved by the addition of lines 4 and 5. What is needed is a tool that will generate specific inductive invariants that

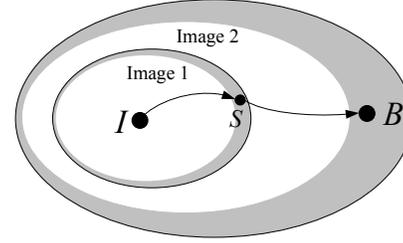


Fig. 2. Interpolation has erroneously reached a bad state.

can demonstrate the unreachability of  $S$  or  $XS$ . Such a tool is described in this paper.

### B. Interpolation

Interpolation is a method that has been found useful for unbounded verification of safety properties. It works by using an over-approximation to the image operation. By applying this image operator iteratively starting from the initial state, either a fixed point is reached or a bad state is encountered. If a fixed point is reached, it is an over-approximation to the set of reachable states such that no bad states are contained in this approximation. The design is verified.

If an approximate image contains a bad state, interpolation may have found a counterexample. However there is in general no way to know if this is a real counterexample or a spurious one. Take for example Figure 2. Two image operations are applied to the initial state  $I$ , and a bad state  $B$  is found in the second image. The shaded region represents the over-approximation inherent in the image operation, and the white shows the true image of the reachable states. We do not know which states of the image lie in the over-approximation, nor do we know if the over-approximated states are reachable. In Figure 2 we have the error trace  $I \rightarrow S \rightarrow B$ , but if either  $S$  or  $B$  lie in the over-approximation and is not truly reachable then the counterexample is spurious.

---

#### Algorithm 2 Modified interpolation.

---

```

1: // Let  $p$  be the property to be proved.
2: set parameters for over-approximate image operator
3:  $\{S\} := I$ 
4: while (1) do
5:    $\{S\}' := \{S\} \cup \text{approxImage}(\{S\})$ 
6:   if  $\{S\}' == \{S\}$  then return “verified”
7:   if  $\exists$  a bad state “near”  $\{S\}$  then
8:     if  $\{S\} == I$  then return “falsified”
9:     if can prove  $s$  unreachable then // See Section IV
10:      record new invariants, goto 7 // See Section IV-D
11:     tighten over-approximation parameters, goto 3
12:   end if
13:    $\{S\} := \{S\}'$ 
14: end while

```

---

Unless specific conditions are met (line 8 of Algorithm

2), the interpolation algorithm has no way of knowing if a counterexample is spurious or true. Therefore, it discards all work up to that point and begins anew with a tighter approximation to the image operator. This restart is costly, and it is a major hot-spot in the performance of the algorithm.

In this work, we have augmented interpolation to call our invariant finding tool as shown in Algorithm 2. On lines 9 and 10, it will find specific inductive invariants that imply that a state along the error trace is unreachable. If invariants are found, the error trace must be spurious, and interpolation is free to proceed without the costly restart on line 11.

#### IV. THE PROOF GRAPH

This section describes the basics of our tool to automatically find and prove useful inductive invariants. A graph structure called the *proof graph* is the core of our method.

##### A. Basic Definitions

The proof graph is a bipartite directed graph with the following node types:

- States in the sequential design. In practice this is a cube of states, but to simplify this discussion, consider only a single state. This constraint will be relaxed in Section V-C.
- Sets of candidate properties. These properties are yet to be proved, but if we can prove them then they are invariants. In this discussion, consider the properties to be chosen from a specific domain. More details on the implemented domain (implications) and details on how to find candidate properties will be given later in Section V-B.

The root of the graph is a single state node. This corresponds to the user-specified state that should be proved unreachable. This root node comes from an outside source – in this work it is a state along the error trace in interpolation. The leaves, i.e. the nodes without outgoing edges, are property sets.

The meaning of the graph lies in its connectivity, specifically in the meanings of edges from states to properties and from properties to states. Being a bipartite graph, there are no other edge types.

*Definition 1 (Edges to Properties):* Let a directed edge from a state  $S$  to a set of properties  $\{P\}$  mean that:

$$\forall p \in \{P\}, \quad S \not\models p$$

That is, all properties  $\{P\}$  fail to hold in  $S$ .

The properties  $\{P\}$  may or may not hold in all reachable states, but if any such  $p \in \{P\}$  can be proved then  $S$  is unreachable. We refer to  $\{P\}$  as a set of *covering properties* for  $S$ .

*Theorem 1 (Proving a State Unreachable):* Let a property  $p$  fail in a state  $S$  ( $S \not\models p$ ). If  $p$  is proved to hold in every reachable state then  $S$  is unreachable.

Therefore the structure  $S \rightarrow \{P\}$  provides a method to show  $S$  unreachable.

*Definition 2 (Edges to States):* Let a directed edge from a set of properties  $\{P\}$  to a state  $S$  mean that:

$$\forall p \in \{P\}, \quad \exists \text{ a successor state } XS \\ \text{such that } (p \models S) \wedge (p \not\models XS)$$

That is, all properties hold in  $S$  but fail in a successor state of  $S$ .

In the structure  $\{P\} \rightarrow S$ ,  $S$  is the reason that the inductive proof of  $\{P\}$  was not successful. In fact,  $S$  is the counterexample to the inductive hypothesis of the proof.

Proving  $S$  to be unreachable is a necessary but not sufficient condition for proving a  $p \in \{P\}$ . Clearly, the proof of  $p \in \{P\}$  cannot succeed if  $S$  may be reachable. Conversely, if  $S$  is known to be unreachable, we have no evidence that a proof of  $p \in \{P\}$  will fail. However, another counterexample  $S'$  may exist.

##### B. An Example

Figure 3 shows an example of a proof graph and how it might evolve over time as our algorithm is run. A sample execution is given here.

- 1) Suppose interpolation reaches a bad state and  $S_0$  is a state on the error trace. We would like to show that  $S_0$  is unreachable. (See Section III-B.)
- 2) Our tool is called to prove that  $S_0$  is unreachable. We set  $S_0$  to be the root of our graph, and through simulation we generate the covering properties  $\{P_0\}$ . In this simulation, we select properties that appear to hold in every reachable state but fail in  $S_0$ . This gives us Graph (1) in Figure 3. (See Section V-B.)
- 3) We attempt to prove the properties  $\{P_0\}$  by simple induction. Suppose that this proof fails, and there are three counterexamples  $S_1$ ,  $S_2$ , and  $S_3$  in the inductive hypothesis. Each counterexample is responsible for disproving a subset of  $\{P_0\}$ , and the proof technique as implemented in [6] will result in these subsets being pair-wise disjoint. We therefore split  $\{P_0\}$  into  $\{P_{0_1}\}$ ,  $\{P_{0_2}\}$ , and  $\{P_{0_3}\}$  such that:
  - $\{P_{0_1}\} \cap \{P_{0_2}\} = \emptyset$ ,  $\{P_{0_1}\} \cap \{P_{0_3}\} = \emptyset$ ,  
 $\{P_{0_2}\} \cap \{P_{0_3}\} = \emptyset$
  - $\{P_{0_1}\} \cup \{P_{0_2}\} \cup \{P_{0_3}\} = \{P_0\}$
  - $\forall j \in \{1, 2, 3\}$ ,  $S_j$  causes the inductive proof of  $\{P_{0_j}\}$  to fail.

Recording this information in the proof graph gives us Graph (2) in the figure. The existence of these counterexamples does not imply that the properties  $\{P\}$  are not true but instead that we need more invariants to prove them. (See Sections III-A and IV-E.)

- 4) Next, cover  $S_1$ ,  $S_2$ , and  $S_3$  with properties, giving us  $\{P_1\}$ ,  $\{P_2\}$ , and  $\{P_3\}$  respectively. These properties provide a way to show that the new states are unreachable. (Like in Step 2, see Section V-B.)

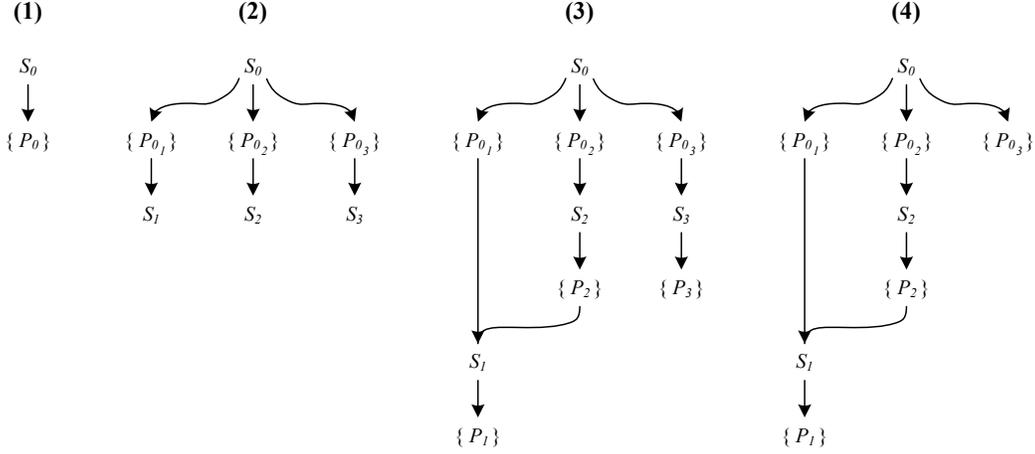


Fig. 3. Sample evolution of a proof graph over time.

- 5) By simulating each new state, we can check to see if it might be responsible for a future failure to prove any of the property sets. Suppose we find that  $S_1$  is a counterexample in the inductive proof of every  $p_2 \in \{P_2\}$ . This induces the edge  $\{P_2\} \rightarrow S_1$ , and the result is Graph (3) in the figure. (See Section V-D.)
- 6) We attempt to prove the properties  $\{P_3\}$ . Suppose we find at least one property to be true for all reachable states. Now  $S_3$  is known to be unreachable, and it can be removed from the proof graph. This gives Graph 4. (See Section IV-D.)
- 7) Attempt the proof of  $\{P_{03}\}$  again. This proof was first attempted in Step 3, but now the reason that the original proof failed,  $S_3$ , is gone and we may re-attempt the proof. Suppose that this time we find that at least one property holds for all reachable states. This implies that  $S_0$ , the root node, is unreachable. At this point, we have achieved our objective and we may return the new invariants to the calling routine, interpolation in this case. (See Section IV-F.)

### C. Selecting Which Properties to Prove

The proof graph in general contains several property set nodes, and the tool must pick one single node to attempt as the next proof. Selecting that node is fairly simple once some basic properties of the proof graph are explored.

*Theorem 2 (Proofs on Leaves Only):* Given the sets of properties  $\{P_0\}$  and  $\{P_1\}$ , a state  $S$ , and the graph structure  $\{P_0\} \rightarrow S \rightarrow \{P_1\}$ . If no  $p_1 \in \{P_1\}$  are proved to hold in every reachable state then  $\forall p_0 \in \{P_0\}$ , it is not possible to prove  $p_0$  by simple induction.

*Proof:* If  $\exists p_1 \in \{P_1\}$  that has been proved, then that would be a guarantee that  $S$  is unreachable. However, because no such proved properties exist, the reachability of  $S$  is unknown. To be conservative, we must allow  $S$  to be a counterexample in the inductive step of the proof of the properties  $\{P_0\}$ . Therefore, the proof will fail  $\forall p_0 \in \{P_0\}$ .

The above theorem defines an order in which the proofs must be attempted. Specifically, if a property node has an outgoing edge to a state then any proof attempt is in vain. In a chain of the graph, only the leaves (nodes without outgoing edges) may be considered as proof candidates. The situation is a bit more complex for a cycle however.

*Theorem 3 (Cycles in The Graph):* Suppose there are property sets  $\{P_0\}, \dots, \{P_n\}$ , unique states  $S_0, \dots, S_n$ , and the cyclic graph structure  $\{P_0\} \rightarrow S_0 \rightarrow \dots \rightarrow \{P_n\} \rightarrow S_n \rightarrow \{P_0\}$ .

If  $\exists j \in \{0, \dots, n\}$  such that  $\forall p_j \in \{P_j\}$ ,  $p_j$  cannot be proved by simple induction  
then  $\forall k \in \{0, \dots, n\}, \forall p_k \in \{P_k\}$ ,  $p_k$  cannot be proved by simple induction

*Proof:* The failure to prove  $\{P_j\}$  results in not being able to prove  $\{P_{(j-1 \bmod n)}\}$  by Theorem 2. This establishes a base case of the inductive proof of this theorem.

Now let  $k \in \{0, \dots, n\}$  and suppose  $\{P_{(k+1 \bmod n)}\}$  cannot be proved. By Theorem 2,  $\{P_k\}$  cannot be proved. Theorem 3 is now proved by induction. ■

Theorem 3 says that in a cycle with  $n$  property set nodes, we must attempt to prove the union of all the property sets at the same time. This simple induction will either successfully prove  $\geq n$  properties or 0 properties because if any properties hold for all reachable states then at least one property in each set must be true.

Cycles must be treated differently from leaves in that the union of the cycle nodes must be proved simultaneously. However, this can be generalized as illustrated in the following example:

- 1) Suppose the current proof graph is that shown in Graph (1) of Figure 4.
- 2) Suppose we find that  $S_0$  can act as the counterexample in the inductive hypothesis for all  $p_1 \in \{P_1\}$ . This induces

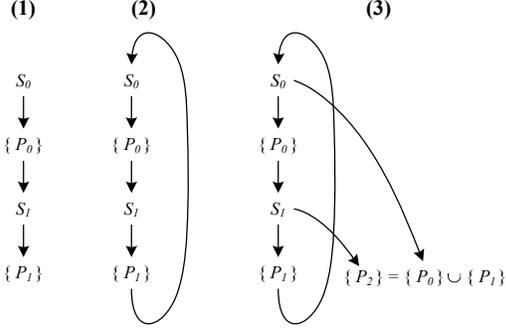


Fig. 4. A cycle has developed in the proof graph.

a cycle in the graph as shown in Graph (2).

- 3) Now create a new leaf node  $\{P_2\} = \{P_0\} \cup \{P_1\}$  and insert it into the proof graph. This records the following information:
  - Both  $\{P_0\}$  and  $\{P_1\}$  must be proved at the same time.
  - A successful proof will imply that both  $S_0$  and  $S_1$  are unreachable.

The updated proof graph is shown in Graph (3).

If cycles are abstracted as illustrated in Figure 4 then a proof of the union of the property sets in the cycle is equivalent to a proof of the new leaf node. After this generalization is made, only the leaves in the proof graph are eligible for an inductive proof.

Suppose a proof graph has multiple leaves, and one leaf must be selected for the simple induction prover. The unique leaf to be given to the induction engine is selected as follows:

- Let  $d(\cdot)$  denote the distance from the root node to a property set node. That is,  $d(\{P\})$  is the number of edges in the shortest directed path from the root node to  $\{P\}$ . Given this metric, the property set  $\{P\}$  which minimizes  $d(\{P\})$  should be selected because it requires fewer inductive proofs to achieve the overall goal, to prove that the root node is unreachable.
- Ties should be broken by selecting the  $\{P\}$  with the greatest cardinality. In the absence of other information about the design, this property set has the greatest chance of having at least one property successfully proved.

#### D. Upon a Successful Proof

Suppose a property set  $\{P\}$  in the proof graph has been selected, given to the simple induction prover, and a property  $p \in \{P\}$  has been proved. This proof is independent from any assumptions, and the proved property is guaranteed to hold for all reachable states. The property is therefore used to simplify all future problems, both simple induction and interpolation. Enforcement of the property can be accomplished by the addition of constraint clauses in each respective SAT instance. This extra clause is maintained throughout the remainder of the execution, effectively utilizing the new invariant in all future problems.

This successful proof also allows the proof graph to be pruned. Theorem 1 implies that all state node parents of  $\{P\}$  are now known to be unreachable. These can be removed from the proof graph, along with any dangling nodes that result. This can create new property set leaves in the graph, enabling the proofs of some property sets to be re-attempted. This happened in Step 6 of the example shown in Section IV-B.

#### E. Upon an Unsuccessful Proof

Suppose in attempting to prove  $\{P\}$ , the simple induction engine fails to prove any of the properties. From Section III-A we know that we can help simple induction by proving that the counterexample state that satisfies the inductive hypothesis is unreachable.

In failing to prove a set of properties, simple induction will produce a set of counterexamples  $\{S_0, S_1, \dots, S_n\}$ . In this case, for each  $p \in \{P\}$ ,  $\exists S_j$  such that  $S_j$  is the reason the inductive proof of  $p$  failed.

To accurately record the relationship between the properties  $\{P\}$  and the states  $\{S_0, S_1, \dots, S_n\}$ ,  $\{P\}$  must be split into  $n$  subsets, one for each of  $\{S_0, S_1, \dots, S_n\}$ . We modify the proof graph by splitting  $\{P\}$ , adding  $\{S_0, S_1, \dots, S_n\}$  along with edges to demonstrate the failed proofs, and lastly we find covering properties  $\{P_j\}$  for each new  $S_j$ . This is illustrated in Steps 3 and 4 of the example in Section IV-B.

One may view each new structure  $S_j \rightarrow \{P_j\}$  as a subgraph rooted at  $S_j$ , the state that the induction engine wants to have shown unreachable. In this way, proving unreachable states for use in the simple induction solver is a sub-problem in the task of proving unreachable states for interpolation.

#### F. Termination Conditions

The above process describes a proof graph that grows as counterexamples are discovered in inductive proofs and shrinks as properties are successfully proved. The proof graph oscillates in size until one of two termination conditions are satisfied:

- If we run out of proof candidates, the overall proof is impossible. This can happen if at some point there are no more leaves in the proof tree. In practice, this means that either the root state node being proved unreachable was in fact reachable or our candidate properties did not provide sufficient information to show this. Absence of leaves is not a guarantee that the root state is reachable.
- If a covering property of the root node is proven to hold for all reachable states then the proof graph algorithm will remove the root from the graph. If the root has been deleted, we can conclude that the root has been proven unreachable and we may stop.

## V. ADVANCED DETAILS

This section is concerned with advanced details in a practical implementation of the method described above. Many details in the previous sections were left abstract, and here we provide the level of detail necessary to implement the proof graph algorithm.

---

**Algorithm 3** The proof graph algorithm.

---

```
1: // Let  $S_0$  be the state to prove unreachable
2: root :=  $S_0$ 
3: cover root with properties
4: while (1) do
5:   if (root ==  $\emptyset$ ) then return “root unreachable”
6:   if (no leaves) then return “root may be reachable”
7:    $\{P\} := \text{selectBestLeaf}()$ 
8:   (proved,  $\{S\}) := \text{simpleInductionProve}(\{P\})$ 
9:   if proved then
10:    delete parents of  $\{P\}$ 
11:   else
12:    for all counterexamples  $s \in \{S\}$  do
13:      make new proof graph node for  $s$ 
14:      cover  $s$  with a new set of properties  $\{P\}$ 
15:      Simulate  $S$ , try to break proofs of all property sets
16:      update proof graph
17:    end for
18:   end if
19: end while
```

---

#### A. Choosing a Property Domain

The proof graph derivation makes use of sets of properties, but the domain of these properties to this point has not been specified. Although an implementation is free to use any domain, ours uses Boolean implications between gates. These were explored in [6]. Similarly, we allow implications to exist between any pair of gates in the design (not limited to the latches). However, we make the restriction that both gates are selected from the same time frame.

Implications were selected because they are expressive and easy to prove. They are much more numerous than alternatives such as node equivalences, yet are more manageable than relations between three nodes. Thus, they are a good trade-off between algorithmic efficiency and expressiveness.

Note that the use of implications as the property domain makes our implementation incomplete. For certain designs,  $\exists$  states  $S_1, S_2$ , such that  $\forall$  sets of implications  $\{P\}$ :

$$\left( S_1 \models \bigwedge_{p \in \{P\}} p \right) \iff \left( S_2 \models \bigwedge_{p \in \{P\}} p \right)$$

That is, implications cannot resolve all pairs of states and so would be unable to demonstrate that exactly one of  $\{S_1, S_2\}$  is reachable because any set of implications would either be satisfied by both states or fail in both states.

#### B. Selecting Covering Properties

In the proof graph algorithm, we often need to find the covering properties  $\{P\}$  for a state  $S$  that have a high probability of holding in all reachable states yet fail to hold in  $S$ . Implications are easy to check with random simulation, and we use simulation extensively to derive covering properties.

Because covering properties will need to be found for many states over the execution of the proof graph algorithm, it

is advantageous to pre-compute a set of implications that have a high probability of holding in all reachable states. We refer to these as *candidate properties*. By constraining random simulation to only simulate reachable states, a set of likely implications can be extracted from a design. This set of implications can be refined through a mix of more random simulation and bounded model checking. In practice, after adequate tuning of the simulation and bounded model checking, this candidate set was of a manageable size for all the benchmarks examined in this paper. See the column “cand. props.” in Table I below.

Using the candidate properties, the covering properties for any particular state  $S$  can be easily derived. The design is simulated using  $S$  as the input vector, and the candidate properties that fail in this simulation are selected as the covering properties for  $S$ .

#### C. State Cubes

Section IV described a proof graph where state nodes consisted of exactly one state in the design. In an implementation this can be relaxed. Instead of being a single state, suppose that a state node represents a cube of states. Whenever possible, a state can be expanded to a cube of states that serves the same purpose, either breaking a simple induction proof, or leading interpolation to an error state. The covering properties can be selected such that any property failing in any state covered by the cube is considered.

Introducing state cubes changes the proof graph theory slightly by introducing a special type of graph cycle.

*Claim 1:* Let  $\{P\}$  be a set of properties and  $S$  a state cube (with  $> 1$  minterms). If there is  $S \rightarrow \{P\} \rightarrow S$  then the proof of  $\{P\}$  is impossible.

This new condition must be checked in the code, and in practice occurs quite frequently. This is another source of graph compression because the property sets that are impossible to prove can be deleted.

#### D. Handling the Large Number of Counterexamples

Inductively proving a set of properties  $\{P\}$  can lead to a large number of counterexamples. This can lead to a blowup in the size of the proof graph, but a few tricks can help to make the size of the proof graph manageable.

**Counterexample Cubes:** The inductive hypothesis of an inductive proof is checked by forming a combinational circuit with a single output that is 1 if and only if the inductive hypothesis is violated for a given set of properties  $\{P\}$ . For every satisfying assignment found for this circuit, we can expand the assignment into a cube by simply discarding primary inputs not appearing on a controlling path in the circuit. This expanded cube is able to violate the inductive hypothesis for a larger subset of  $\{P\}$  because in practice the counterexamples found are usually clustered. By using counterexample cubes, the total number of counterexamples found is reduced.

**Proof Graph Bounding:** In the proof graph, the primary goal is to prove the root state to be unreachable. To do

TABLE I  
PERFORMANCE ON A SAMPLING OF HARD ACADEMIC PROBLEMS

Design	Design Properties			Standard Interpolation			Interpolation + Proof Graph				
	And Gates	Latches	Prop. Type	MB	Sec.	Refines	MB	Sec.	Refines	Cand. Props.	Props. Proved
cmu_dme1_B	236	61	Unknown	2484	7200	5	2487	7200	5	390	0
cmu_dme2_B	296	63	Unknown	2507	7200	7	2674	7200	7	604	0
eijk_S1423_S	902	159	True	2481	7200	1	139	77.93	0	10078	2400
eijk_S208_S	109	22	True	2451	7200	4	38	60.62	0	1668	454
eijk_S208c_S	111	23	True	2469	7200	7	30	59.04	0	1864	660
eijk_S382_S	230	57	True	2480	7200	5	228	102.17	0	23144	4176
eijk_S420_S	243	50	True	2500	7200	7	148	191.27	0	11000	2250
eijk_S444_S	240	57	True	2491	7200	5	224	507.37	0	38530	28972
eijk_S838_S	480	106	True	2570	7200	2	1199	370.63	0	152734	27308
eijk_bs1512_S	866	158	Unknown	2471	7200	0	2475	7200	0	46108	60
eijk_bs3271_S	1841	305	Unknown	1822	7200	1	2514	7200	1	6544	772
irst_dme4_B	593	124	Unknown	2515	7200	4	2558	7200	4	1894	0
irst_dme5_B	790	165	Unknown	2562	7200	5	528	7200	4	16590	0
irst_dme6_B	1181	245	Unknown	2564	7200	5	440	7200	0	126850	0
nusmv_brp_B	375	52	Unknown	2467	7200	1	805	7200	1	9140	1128
nusmv_queue_B	1310	84	True	2459	7200	1	95	151.78	0	3690	480
nusmv_reactor_6_C	903	76	True	2478	7200	7	52	140.66	0	6228	482
vis_bakery_E	284	25	Unknown	2454	7200	3	2582	7203.06	5	4018	626

this, only one path of successful proofs leading back to the root node is needed. This means that most of the counterexamples appearing in the proof graph are not needed to show that the root state is unreachable. Harnessing this idea, the implementation bounds the number of counterexample nodes appearing in the proof graph. If adding a new counterexample would cause the number of counterexample nodes to exceed this bound then the new counterexample is ignored. By using a sufficiently high counterexample node bound we can dramatically reduce the size of the proof graph and minimally impact the invariants found by our algorithm.

**Simulated Inductive Proofs:** Often a counterexample encountered previously can serve to break the inductive proof of a newly derived set of covering properties. Similarly, new counterexamples from the simple induction engine can often serve to break the proof of many sets of covering properties, more than just what the induction engine is currently processing. To detect these cases, random simulation is used to determine if a specific counterexample can satisfy the inductive hypothesis of a specific set of covering properties. This encourages re-use of old counterexamples rather than forcing new counterexamples to be derived, and in practice it works quite well. For an example, see Step 5 of Section IV-B.

## VI. EXPERIMENTAL RESULTS

For this work, we implemented two C++ plugins for the ABC Logic Synthesis and Verification System [10]. The first plugin implements the interpolation algorithm as described in [4], and the second plugin implements the invariant discovery method proposed in this paper. The plugins are interfaced as described in Algorithm 2 to provide inductive invariants for aiding interpolation.

We experimented with a suite of 154 academic designs that had been annotated with safety properties [12]. The designs ranged in size from 10 to 689 latches. After the designs

were combinationally synthesized into And-Inverter Graphs, they had between 43 and 3716 And nodes. Each design in this benchmark suite contains a single safety property, which include 95 true properties, 34 false properties, and 25 properties of unknown nature.

The technique described in this paper can greatly speed-up model checking, but also it imposes some overhead to find and prove inductive invariants. The algorithm is best suited to run as an option in a verification package that can be invoked after more conventional methods have been exhausted. To emulate this type of flow, we attempted to verify all 154 benchmarks with standard interpolation. 132 finished in less than 10 minutes, and 18 failed to verify in 2 hours. It is on these 18 that we then applied our method.

Table I shows these 18 benchmarks on which interpolation times-out after 2 hours. As discussed in Section III-B, the most expensive part of the interpolation algorithm is the model refinement. The number of refinements done in the standard interpolation algorithm is shown in the table, and in some cases this number is quite high.

If specific inductive invariants can be found, then refinement can be avoided. In the last part of the table, we show the statistics for an implementation of interpolation that utilizes the proof graph. Whenever interpolation reaches a bad state, it will find and prove appropriate inductive invariants. In half of the designs, proving a small number of properties was sufficient to allow all model refinement steps to be skipped. Runtime was dramatically improved in those cases, and the inductive invariants proved to be the difference between a time-out and a successful verification run.

Interestingly, no false properties are present in Table I. The technique presented here favors true properties because for these, any trace into a bad state must contain unreachable states and so there is an opportunity to find invariants to cover those states. With a falsifiable property, the error trace will only contain reachable states and the proof graph method will waste resources attempting to show that these states are

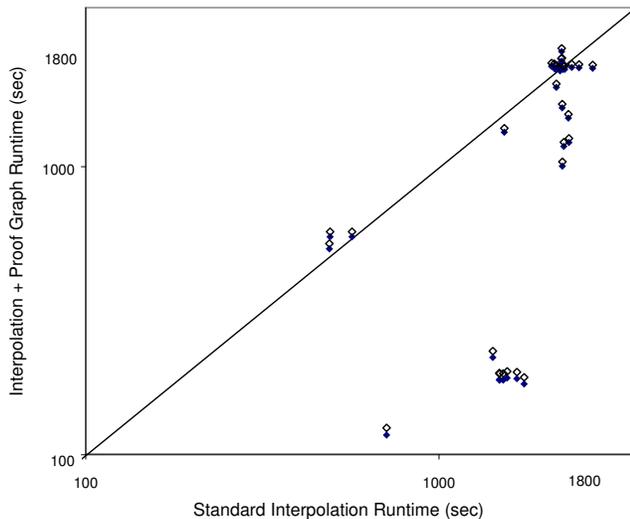


Fig. 5. Performance On A Sampling of Hard Industrial Problems

unreachable. Consequently, if there are indeed false properties in this subset of the benchmark suite, these probably appear as time-out cases.

Further experimentation was done using 43 industrial designs. Each contains multiple properties which are to be proven one-by-one. These designs were chosen because they are the hardest available, each having runtimes in excess of 8 minutes when processed with standard interpolation.

Figure 5 shows a scatter plot comparison of standard interpolation versus interpolation aided with the proof graph. Each verification run was given a time-out of 30 minutes, and the proof graph successfully prevented 5 of the designs from exceeding the time limit. Even on designs that did not time-out with standard interpolation, using the proof graph significantly helped the runtime.

Future work will include trying this technique on more industrial benchmarks, specifically ones that are harder for standard interpolation.

## VII. CONCLUSION

This paper outlined a method to automatically extract and prove inductive invariants. These invariants are used to show that a single, application-specified state is unreachable, and in this way only invariants that are immediately helpful to the problem at hand are found. This is accomplished by use of the proof graph, a structure whose theory and application were thoroughly explored here.

The proof graph shows that there is a fine relationship

between different properties of a design. Some can be independently proved by simple induction. Others require that disjoint sets of properties be proved in a particular order. Some properties can only be proved in conjunction with other properties. Proving local properties with simple induction was also explored in our previous work [6], but this work provides us with a more complete understanding of this proof process. Specifically, the proof graph is an effective tool to capture all of the dependencies between the inductive proofs of different property sets.

This technique was applied to the interpolation algorithm, and the feasibility of modifying this algorithm to request and utilize specific invariants was demonstrated. It should be possible to incorporate our proof graph algorithm into any tool that needs specific invariants, and thus the methods presented in this work could have widespread application.

## ACKNOWLEDGMENTS

The authors would like to acknowledge the Semiconductor Research Corporation (SRC) for their support through contracts 1361.001 and 1444.001. Also, we would like to thank Sanjit Seshia for his guidance in the class where this project began.

## REFERENCES

- [1] G. Hasteer, "Efficient Equivalence Checking in a Modular Design Environment," PhD Thesis from University of Illinois at Urbana-Champaign, 1998.
- [2] M. Prasad, A. Biere, A. Gupta, "A Survey of Recent Advances in SAT-Based Formal Verification," in *Software Tools for Technology Transfer (STTT)*, Vol. 7, No. 2, 2005, pp. 156-173
- [3] J. Baumgartner, "Integrating FV Into Main-Stream Verification: The IBM Experience," Tutorial Given at *FMCAD 2006*
- [4] K.L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. CAV 2003*, pp. 1-13.
- [5] N. Amla, X. Du, A. Kuehlmann, R.P. Kurshan, and K.L. McMillan, "An Analysis of SAT-based Model Checking Techniques in an Industrial Environment," in *CHARME 2005*.
- [6] M.L. Case, A. Mishchenko, and R.K. Brayton, "Inductively Finding a Reachable State Space Over-Approximation," in *IWLS 2006*.
- [7] C. A. J. van Eijk, "Sequential Equivalence Checking without State Space Traversal," In *Proceedings of DATE*, February 1998.
- [8] P. Bjesse and K. Claessen, "SAT-based Verification without State Space Traversal," in *FMCAD*, 2000.
- [9] M. Wedler, D. Stoffel, and W. Kunz, "Exploiting state encoding for invariant generation in induction-based property checking," in *ASP-DAC*, Jan. 2004.
- [10] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [11] R.K. Brayton, "The real reason why McMillan's construction is an interpolant." ERL Technical Report, EECS Dept., UC Berkeley.
- [12] Property Checking Benchmark Suite, <http://www.cs.chalmers.se/~een/Tip/>
- [13] Niklas Een, Niklas Sorensson, MiniSat. <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/>
- [14] F. Lu, M. K. Iyer, G. Parthasarathy, and K. T. Cheng, "An efficient sequential SAT solver with improved search strategies," in *DATE 2005*.

# Checking Safety by Inductive Generalization of Counterexamples to Induction

Aaron R. Bradley and Zohar Manna

Computer Science Department  
Stanford University  
Stanford, CA 94305-9045  
Email: {arbrad, manna}@cs.stanford.edu

**Abstract**—Scaling verification to large circuits requires some form of abstraction relative to the asserted property. We describe a safety analysis of finite-state systems that generalizes from counterexamples to the inductiveness of the safety specification to inductive invariants. It thus abstracts the system’s state space relative to the property. The analysis either strengthens a safety specification to be inductive or discovers a counterexample to its correctness. The analysis is easily made parallel. We provide experimental data showing how the analysis time decreases with the number of processes on several hard problems.

## I. INTRODUCTION

We describe a safety analysis for finite-state systems that incrementally strengthens a correct specification to be inductive. Each iteration of the analysis chooses a *counterexample to the inductiveness of the specification*: a state that explains why the specification is not yet inductive. It then generalizes from the assumption that this state is unreachable to an inductive invariant that proves that many states, including the counterexample to induction, are unreachable. Thus, through directed invariant generation, it accomplishes an abstraction of the state space of the system relative to the specification.

The main idea of our analysis is the following. Suppose the given safety property is not inductive. Then there exists a *counterexample to induction* (CTI): a state that can lead to a violation of the property. The analysis attempts to generate a single inductive invariant in the form of a clause that eliminates this state and many other states as well. If it succeeds, the generated inductive clause strengthens the property. Otherwise, the given property is extended to assert that the CTI state is also unreachable. After many iterations of this process, either the incrementally strengthened property becomes inductive or a counterexample trace is found. Section III presents the analysis in detail, and Section VII discusses the analysis more generally.

The application of induction as the basis for generalization is the distinguishing feature of this analysis. From one CTI, the analysis generalizes to an inductive clause that proves that many states are unreachable: the CTI state itself, all the states

This research was supported in part by NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, by ARO grant DAAD19-01-1-0723, and by NAVY/ONR contract N00014-03-1-0939. The first author was additionally supported by a Sang Samuel Wang Stanford Graduate Fellowship.

that can reach it, and *many other states that cannot reach it*. This generalization beyond the states that can reach the CTI contrasts with standard preimage computation.

The success of our approach depends on two points. First, we require a fast method of generating inductive clauses. We address this challenge in Section IV. Second, small inductive clauses must exist in practice. The empirical evidence of Section V suggests that they do. In Section VI, we discuss related techniques and the potential for hybrid approaches.

## II. PRELIMINARIES

### A. Propositional Logic

Let us review a few useful notations and definitions of propositional logic. A *literal*  $\ell$  is a propositional variable  $x$  or its negation  $\neg x$ . A *clause*  $c$  is a disjunction of literals. The size  $|c|$  of clause  $c$  is its number of literals. A subclause  $d \sqsubseteq c$  is a disjunction of a subset of literals of  $c$ . A formula in *conjunctive normal form* (CNF) is a conjunction of clauses.

We write  $\varphi[\bar{x}]$  to indicate that the formula  $\varphi$  has variables  $\bar{x} = \{x_1, \dots, x_n\}$ . An *assignment*  $s$  associates a truth value  $\{\text{true}, \text{false}\}$  with each variable  $x_i$  of  $\bar{x}$ .  $\varphi[s]$  is the truth value of  $\varphi$  on assignment  $s$ , and  $s$  is a *satisfying assignment* of  $\varphi$  if  $\varphi[s]$  is true. A *partial assignment*  $t$  need not assign values to every variable.

Finally, a formula  $\varphi$  *implies* a formula  $\psi$  if every satisfying assignment (that assigns a truth value to every variable of  $\varphi$  and  $\psi$ ) of  $\varphi$  also satisfies  $\psi$ . In this case, we say that the implication  $\varphi \Rightarrow \psi$  holds. The implication  $\varphi \Rightarrow \psi$  holds iff (if and only if) the formula  $\varphi \wedge \neg\psi$  is unsatisfiable.

### B. Transition Systems

We model finite-state systems as Boolean transition systems.

**Definition 1 (Boolean Transition System)** A *Boolean transition system*  $\mathcal{S}$ :  $\langle \bar{x}, \theta, \rho \rangle$  has three components:

- a set of propositional variables  $\bar{x} = \{x_1, \dots, x_n\}$ , which hold the state of the system;
- an *initial condition*, a propositional formula  $\theta[\bar{x}]$ , which describes in which states the system can start;
- and a *transition relation*, a propositional formula  $\rho[\bar{x}, \bar{x}']$ , which describes how the system evolves in each step of execution.

In  $\rho[\bar{x}, \bar{x}']$ , primed variables  $\bar{x}'$  represent the values of the variables  $\bar{x}$  in the next state.

The semantics of a finite-state system are given by its set of computations.

**Definition 2 (State & Computation)** A *state*  $s$  of Boolean transition system  $\mathcal{S}$  is an assignment of the variables  $\bar{x}$ . A *computation*  $\sigma : s_0, s_1, s_2, \dots$  is a sequence of states such that

- $s_0$  satisfies the initial condition:  $\theta[s_0]$  is true;
- and for each  $i \geq 0$ ,  $s_i$  and  $s_{i+1}$  are related by  $\rho$ :  $\rho[s_i, s_{i+1}]$  is true.

A state of  $\mathcal{S}$  is *reachable* if it is in some computation of  $\mathcal{S}$ . We are interested in determining for a given system  $\mathcal{S}$  and formula  $\varphi$  if every reachable state of  $\mathcal{S}$  satisfies  $\varphi$ . If so, then  $\varphi$  is  *$\mathcal{S}$ -invariant*, or is an *invariant* of  $\mathcal{S}$ . If not, then there exists a computation with some state  $s$  that falsifies  $\varphi$ ; the prefix of the computation up to  $s$  is a *counterexample trace*.

Rather than examining each of the possibly infinitely many computations of  $\mathcal{S}$ , however, we apply induction to determine if  $\varphi$  is  $\mathcal{S}$ -invariant.

**Definition 3 (Inductive Invariant)** A formula  $\varphi$  is  *$\mathcal{S}$ -inductive* if

- it holds initially:  $\theta \Rightarrow \varphi$ , (initiation)
- and it is preserved by  $\rho$ :  $\varphi \wedge \rho \Rightarrow \varphi'$ . (consecution)

In the latter formula,  $\varphi'$  abbreviates  $\varphi[\bar{x}']$ : each variable is replaced by its corresponding primed variable.

A formula  $\varphi$  is  *$\mathcal{S}$ -inductive relative to a  $\mathcal{S}$ -inductive formula  $\psi$*  if  $\theta \Rightarrow \varphi$  as before, and  $\psi \wedge \varphi \wedge \rho \Rightarrow \varphi'$ .

The two implications — the base case and the inductive case, respectively — are the *verification conditions* for  $\mathcal{S}$  and  $\varphi$ . When  $\mathcal{S}$  is obvious from the context, we omit it from  $\mathcal{S}$ -inductive and  $\mathcal{S}$ -invariant.

### III. FINITE-STATE INDUCTIVE STRENGTHENING

Given transition system  $\mathcal{S} : \langle \bar{x}, \theta, \rho \rangle$  and specification formula  $\Pi$ , is  $\Pi$   $\mathcal{S}$ -invariant? Proving that  $\Pi$  is inductive answers the question affirmatively. However,  $\Pi$  is often invariant yet not inductive. A standard approach in this case is to find a formula  $\chi$  such that  $\Pi \wedge \chi$  is inductive;  $\chi$  is called a *strengthening assertion* [1]. There are many approaches to finding a strengthening assertion for finite-state systems (see Section VI). We describe an approach based on generating many clauses, each of which is inductive relative to the previously-generated clauses.

#### A. Inductive Clauses

First, let us consider how to find a single clause that is inductive relative to some formula  $\psi$ . Later, we show how to chain the discovery of such clauses together to decide whether a system  $\mathcal{S}$  meets its specification  $\Pi$ . Our presentation is self-contained but follows the ideas of abstract interpretation [2].

Consider an arbitrary clause  $c$  that need not be inductive. It induces the *subclause lattice*  $L_c : \langle 2^c, \sqcap, \sqcup, \sqsubseteq \rangle$  in which

- the elements  $2^c$  are the subclauses of  $c$ ;
- the elements are ordered by the subclause relation  $\sqsubseteq$ : in particular, the top element is  $c$  itself, and the bottom element is the formula false (“false”);
- the join operator  $\sqcup$  is simply disjunction;
- the meet operator  $\sqcap$  is defined as follows:  $c_1 \sqcap c_2$  is the disjunction of the literals common to  $c_1$  and  $c_2$ .

$L_c$  is complete; by the Knaster-Tarski theorem [3], [4], every monotone function on  $L_c$  has a least and a greatest fixpoint.

Consider the monotone nonincreasing function  $\text{down}(L_c, d)$  that, given the subclause lattice  $L_c$  and the clause  $d \in 2^c$ , returns the (unique) largest subclause  $e \sqsubseteq d$  such that the implication  $\psi \wedge e \wedge \rho \Rightarrow d'$  holds. In other words, it returns the best under-approximation in  $L_c$  to the weakest precondition of  $d$ . If the greatest fixpoint  $\bar{c}$  of  $\text{down}(L_c, c)$  satisfies the implication  $\theta \Rightarrow \bar{c}$  of initiation, then it is the largest subclause of  $c$  that is inductive relative to  $\psi$ . Section IV-A describes how to find  $\bar{c}$  with a number of satisfiability queries linear in  $|c|$ .

Large inductive clauses are undesirable, however, because they provide less information than smaller clauses: they are satisfied by more states. We want to find a *minimal inductive subclause* of  $c$ , an inductive subclause that does not itself contain any strict subclause that is inductive. Therefore, we examine the least fixpoint of a monotone *nondecreasing* function on *inductive* subclause lattices, which are lattices whose top elements are inductive. Constructing an inductive subclause lattice requires first computing the greatest fixpoint of  $\text{down}$  on a larger subclause lattice.

To that end, consider the (nondeterministic) function  $\text{implicate}(\varphi, c)$  that, given formula  $\varphi$  and clause  $c$ , returns a minimal subclause  $d \sqsubseteq c$  such that  $\varphi \Rightarrow d$  if such a clause exists, and returns  $\top$  (“true”) otherwise. This minimal subclause  $d$  is known as a *prime implicate* [5], [6]. There can be exponentially many such minimal subclauses since the CNF representation of a formula can be exponentially large. But there may not be any prime implicate if  $\varphi \not\Rightarrow c$ . Section IV-B presents an optimal implementation of  $\text{implicate}$ .

Using  $\text{implicate}$ , we can find a subclause of  $c$  that best approximates  $\theta$ :  $b : \text{implicate}(\theta, c)$ . Consider the subclause *sublattice*  $L_{b,c}$  of  $L_c$  that has top element  $c$  and bottom element  $b$ . Consider also the operation  $\text{up}(L_{b,c}, d)$  that, for element  $d$  of  $L_{b,c}$ , returns a minimal subclause  $e \sqsubseteq c$  such that the implication  $\psi \wedge d \wedge \rho \Rightarrow e'$  holds. In other words, it computes  $e' : \text{implicate}(\psi \wedge d \wedge \rho, c')$ , a best over-approximation in  $L_{b,c}$  of the strongest postcondition of  $d$ .

The operation  $\text{up}$  is a function on  $L_{b,c}$  — it maps an element of  $L_{b,c}$  to an element of  $L_{b,c}$  — precisely when the top element  $c$  is inductive. In this case, its least fixpoint  $\bar{c}$  is an inductive subclause of  $c$  that is small in practice. However, it is not necessarily a minimal inductive subclause, as different deterministic instantiations of  $\text{implicate}$  result in different least fixpoints, some of which may be strict subclauses of others.

Now for a given clause  $c$ , compute the greatest fixpoint of  $\text{down}$  on  $L_c$  to discover inductive clause  $\bar{c}$  and its corresponding inductive sublattice  $L_{\bar{c}}$ . Compute  $b : \text{implicate}(\theta, c)$  to identify the inductive sublattice  $L_{b,\bar{c}}$  whose bottom element

over-approximates  $\theta$ . Finally, compute the least fixpoint of  $\text{up}$  on  $L_{b,\bar{c}}$  to find a small inductive subclause  $\bar{d}$  of  $c$ . In practice,  $\bar{d}$  is small but need not be minimal.

A brute-force recursive technique finds a minimal inductive subclause. First apply the procedure described above to  $c$  to find  $\bar{d}$ . Then recursively treat each immediate strict subclause of  $\bar{d}$ , of which there are  $|\bar{d}|$ . A clause  $\bar{d}$  is a minimal inductive subclause of  $c$  precisely when each of these recursive calls fails to find an inductive strict subclause of  $\bar{d}$ . We call this procedure  $\text{MIC}(\mathcal{S}, \psi, c)$ . It returns a minimal subclause of  $c$  that is  $\mathcal{S}$ -inductive relative to  $\psi$ , or  $\top$  if no such clause exists.

### B. FSIS: Generalizing from Counterexamples to Induction

Having developed the algorithm  $\text{MIC}$  to discover a minimal inductive subclause of a given clause  $c$ , the remaining consideration is which clause  $c$  to provide it. We use negations of *counterexamples to induction* (CTIs):  $\Pi$ -states that can lead to  $\neg\Pi$ -states. This choice guides  $\text{MIC}$  to discover inductive clauses that are relevant for proving that  $\Pi$  is  $\mathcal{S}$ -invariant, implicitly abstracting the state-space of  $\mathcal{S}$  relative to  $\Pi$ . Hence, rather than finding a CNF representation of the exact set of reachable states of  $\mathcal{S}$ , we expect to find a much smaller formula  $\chi$  such that  $\Pi \wedge \chi$  is inductive and represents a larger set of states that all satisfy  $\Pi$ .

Suppose that  $\chi$  is a conjunction of previously-generated clauses that are  $\mathcal{S}$ -inductive relative to  $\Pi$ , but  $\Pi \wedge \chi$  is not  $\mathcal{S}$ -inductive. Why is  $\Pi \wedge \chi$  not inductive? One possibility is that initiation fails: the formula  $\theta \wedge \neg\Pi$  is satisfied by some state  $s$ . In this case,  $\Pi$  is not  $\mathcal{S}$ -invariant.

Another possibility is that consecution fails: the formula  $\Pi \wedge \chi \wedge \rho \wedge \neg\Pi'$  is satisfied by some pair of states  $(s, s')$ . That is, it is possible for  $\mathcal{S}$  to transition from state  $s$ , which satisfies  $\Pi \wedge \chi$ , to state  $s'$ , which satisfies  $\neg\Pi'$  and thus violates the specification  $\Pi$ . State  $s$  is a CTI.

Since  $s$  is an assignment of truth values to variables of  $\mathcal{S}$ , it can be viewed as a conjunction of literals: if  $s$  assigns  $x_i$  to be true, the conjunction contains  $x_i$ ; if  $s$  assigns  $x_i$  to be false, the conjunction contains  $\neg x_i$ . Call this conjunction  $\hat{s}$ . Now, noticing that  $\neg\hat{s}$  is a clause, compute  $\text{MIC}(\mathcal{S}, \Pi \wedge \chi, \neg\hat{s})$ . If  $\text{MIC}$  returns  $\top$  because  $\hat{s}$  does not contain a subclause that is inductive relative to  $\Pi \wedge \chi$ , then update  $\Pi$  to indicate that proving that state  $s$  is unreachable is a subgoal of proving the invariance of the original  $\Pi$ :  $\Pi := \Pi \wedge \neg\hat{s}$ . Otherwise,  $\text{MIC}$  returns  $\bar{c}$ , an inductive generalization of  $\neg\hat{s}$ :  $\bar{c}$  is inductive relative to  $\Pi \wedge \chi$  and excludes state  $s$  and many other states. Update  $\chi$  accordingly:  $\chi := \chi \wedge \bar{c}$ .

Eventually, either  $\Pi$  grows so that  $\theta \wedge \neg\Pi$  is satisfiable, disproving the specification, or  $\Pi \wedge \chi$  becomes inductive. For  $\chi$  is always inductive relative to  $\Pi$ ; and if the formula  $\Pi \wedge \chi \wedge \rho \wedge \neg\Pi'$  is unsatisfiable, then  $\Pi$  is inductive relative to  $\chi$ .

Which states does an inductive generalization of  $\neg\hat{s}$  exclude? It clearly excludes state  $s$ . It also excludes all states that can reach  $s$ , for it would not be inductive otherwise. However, even the largest inductive subclause of  $\neg\hat{s}$  excludes all states that can reach  $s$ , while  $\text{MIC}$  discovers significantly smaller inductive subclauses in practice. Hence,  $\text{MIC}$  generalizes the

argument that  $s$  and those states that can reach  $s$  are unreachable to prove that many other states are also unreachable. This generalization contrasts with methods that compute the preimage of a CTI [7]–[9].

This analysis is naturally made parallel. By simply using a randomized decision procedure to obtain the CTIs, each process is likely to analyze a different part of the state-space. Processes need only communicate discovered inductive clauses and — depending on implementation choices (see Section V) — CTIs that do not yield inductive clauses.

We call this procedure  $\text{FSIS}(\mathcal{S}, \Pi)$ , for *finite-state inductive strengthening*. It returns an inductive strengthening of  $\Pi$  if  $\Pi$  is  $\mathcal{S}$ -invariant; otherwise, it extracts from the subgoals conjoined to  $\Pi$  a counterexample trace.

### C. One-Step Cone of Influence

One need not consider a full assignment  $s$  as a CTI. Computing a *one-step cone of influence* (COI) — the variables that can possibly impact the truth value of  $\Pi$  in the next state — yields a partial assignment  $t$  that describes a set of states, rather than just a single state  $s$ , that can lead to a violation of  $\Pi$ . Applying  $\text{MIC}$  to the resulting clause  $\neg\hat{t}$  focuses it on excluding all of these states, rather than proving that just the state  $s$  is unreachable for a reason that is unrelated to why the other states are unreachable.

## IV. ALGORITHMS

This section develops the algorithms introduced in Section III-A to discover minimal inductive subclauses.

### A. Computing the Largest Inductive Subclause

Recall that the monotone nonincreasing function  $\text{down}(L_c, d)$  computes the largest subclause  $e \sqsubseteq d$  such that the implication  $\psi \wedge e \wedge \rho \Rightarrow d'$  holds. A straightforward method of computing the greatest fixpoint of  $\text{down}$  in  $L_c$  — which iteratively computes under-approximations to the weakest precondition — can require  $\Omega(|c|^2)$  satisfiability queries. For systems with many variables, this quadratic cost is prohibitively expensive.

We describe a method that requires a linear number of queries. Consider checking if the implication  $\psi \wedge c \wedge \rho \Rightarrow c'$  holds. If it does, and if the implication  $\theta \Rightarrow c$  of initiation also holds, then  $c$  is inductive relative to  $\psi$ . If it does not, then the formula  $\psi \wedge c \wedge \rho \wedge \neg c'$  is satisfied by some assignment  $(s, s')$  to the unprimed and primed variables. Let  $\hat{s}$  be the conjunction of literals corresponding to  $s$ ; and let  $\neg\hat{t}$  be the best over-approximation to  $\neg\hat{s}$  in  $L_c$ , which is the largest clause with literals common to  $c$  and  $\neg\hat{s}$ . Then compute the new clause  $d : c \sqcap \neg\hat{t}$ . In other words,  $d$  has the literals common to  $c$  and  $\neg\hat{s}$ . Now recurse on  $d$ .

If at any point during the computation, initiation does not hold, then report failure.

This algorithm, which we call  $\text{LIC}(L_c, c)$ , computes the largest inductive subclause of the given clause  $c$ .

**Proposition 1 (Largest Inductive Subclause)** *The fixpoint of the iteration sequence computed by  $\text{LIC}(L_c, c)$  is the*

largest subclause of  $c$  that satisfies consecution. If it also satisfies initiation, then it is the largest inductive subclause of  $c$ . Finding it requires solving at most  $O(|c|)$  satisfiability queries.

*Proof:* Let the computed sequence be  $c_0 = c, c_1, \dots, c_k$ , where the fixpoint  $c_k$  satisfies consecution. Notice that for each  $i > 0$ ,  $c_i \sqsubseteq c_{i-1}$  by construction. Suppose that  $e \sqsubseteq c$  also satisfies consecution, yet it is not a subclause of  $c_k$ . We derive a contradiction.

Consider position  $i$  at which  $e \sqsubseteq c_i$  but  $e \not\sqsubseteq c_{i+1}$ ; such a position must occur by the existence of  $e$ . Now partition  $c_i$  into two clauses,  $e \vee f$ ;  $f$  contains the literals of  $c_i$  that are not literals of  $e$ . Since consecution does not yet hold for  $c_i$ , the formula  $\psi \wedge (e \vee f) \wedge \rho \wedge \neg(e' \vee f')$  is satisfiable. Case splitting, one of the following two formulae is satisfiable:

$$\psi \wedge e \wedge \rho \wedge \neg e' \wedge \neg f' \quad (1)$$

$$\psi \wedge \neg e \wedge f \wedge \rho \wedge \neg e' \wedge \neg f' \quad (2)$$

Formula (1) is unsatisfiable because  $e$  satisfies consecution by assumption. Therefore, formula (2) is satisfied by some assignment  $(s, s')$ . Now, because  $\neg e[s]$  evaluates to true, we know that  $e \sqsubseteq \neg \hat{s}$  (where  $\hat{s}$  is the conjunction of literals corresponding to assignment  $s$ ); but then  $e \sqsubseteq c_{i+1} = c_i \sqcap \neg \hat{s}$ , a contradiction.

The linear bound on the number of satisfiability queries follows from the observation that each iteration (other than the final one) must drop at least one literal of  $c$ . ■

We thus have an algorithm for computing the largest inductive subclause of a given clause with only a linear number of satisfiability queries.

In practice, during one execution of MIC, the clauses that are found not to contain inductive clauses should be cached to preclude the future exploration of its subclauses.

### B. Computing Prime Implicates

Recall that the monotone nondecreasing function  $\text{up}(L_c, d)$  computes a minimal subclause  $e \sqsubseteq c$  such that the implication  $\psi \wedge d \wedge \rho \Rightarrow e'$  holds. As explained in Section III-A, the crucial part of implementing  $\text{up}$  is implementing an algorithm for finding minimal implicates:  $\text{implicate}(\varphi, c)$  should return a minimal subclause of  $c$  such that  $\varphi \Rightarrow c$  holds, or  $\top$  if no such subclause exists. We focus on  $\text{implicate}$  in this section.

In fact, we consider a more general problem. Consider a set of objects  $S$  and a predicate  $p : 2^S \mapsto \{\text{true}, \text{false}\}$  that is monotone on  $S$ : if  $p(S_0)$  is true and  $S_0 \subseteq S_1 \subseteq S$ , then also  $p(S_1)$  is true. We assume that  $p(S)$  is true; this assumption can be checked with one preliminary query. The problem is to find a minimal subset  $\bar{S} \subseteq S$  that satisfies  $p$ :  $p(\bar{S})$ .

The correspondence between this general problem and  $\text{implicate}(\varphi, c)$  is direct: let  $S$  be the set of literals of  $c$  and  $p$  be the predicate that is true for  $S_0 \subseteq S$  precisely when  $\varphi \Rightarrow \bigvee S_0$ , where  $\bigvee S_0$  is the disjunction of the literals of  $S_0$ .

A straightforward and well-known algorithm for finding a minimal satisfying subset of  $S$  requires a linear number of queries to  $p$ . Drop an element of the given set. If the remaining

```

let rec min p S0 = function
  | [] → S0
  | h :: t → if p(S0 ∪ t)
              then min p S0 t
              else min p (h :: S0) t
let minimal p S = min p [] S

```

Fig. 1. Linear-time minimal

```

let rec split (l, r) = function
  | [] → (l, r)
  | h :: [] → (h :: l, r)
  | h1 :: h2 :: t → split (h1 :: l, h2 :: r) t
let split S0 = split ([], []) S0
let rec min p sup S0 =
  if |S0| = 1
  then S0
  else let l0, r0 = split S0 in
        if p(sup ∪ l0)
        then min p sup l0
        else if p(sup ∪ r0)
              then min p sup r0
              else let l = min p (sup ∪ r0) l0 in
                    let r = min p (sup ∪ l) r0 in
                    l ∪ r
let minimal p S = min p [] S

```

Fig. 2. Optimal minimal

set satisfies  $p$ , recurse on it; otherwise, recurse on the given set, remembering never to drop that element again.

Figure 1 describes this algorithm precisely using an O’Caml-like language. It treats sets as lists.  $S_0$  contains the required elements of  $S$  that have already been examined; if there are not any remaining elements, return  $S_0$ . Otherwise, the remaining elements consist of  $h :: t$  — a distinguished element  $h$  (the “head”) and the other elements  $t$  (the “tail”). If  $p(S_0 \cup t)$  is true,  $h$  is unnecessary; otherwise, it is necessary, so add it to  $S_0$ . We provide these details to prepare the reader to understand an algorithm that makes the optimal number of queries to  $p$ .

We can do exponentially better than always making a linear number of queries to  $p$ . Suppose we are given two disjoint sets, the “support” set  $\text{sup}$  and the set  $S_0$ , such that  $p(\text{sup} \cup S_0)$  holds but  $p(\text{sup})$  does not hold. We want to find a minimal subset  $\bar{S} \subseteq S_0$  such that  $p(\text{sup} \cup \bar{S})$  holds. If  $S_0$  has just one element, then that one element is definitely necessary, so return it. Otherwise, split  $S_0$  into two disjoint subsets  $\ell_0$  and  $r_0$  with roughly half the elements each (see Figure 2 for a precise description of  $\text{split}$ ). Now if  $p(\text{sup} \cup \ell_0)$  is true, immediately recurse on  $\ell_0$ , using  $\text{sup}$  again as the support set. If not, but  $p(\text{sup} \cup r_0)$  is true, then recurse on  $r_0$ , using  $\text{sup}$  again as the support set.

The interesting case occurs when neither  $p(\text{sup} \cup \ell_0)$  nor  $p(\text{sup} \cup r_0)$  hold: in this case, elements are required from both  $\ell_0$  and  $r_0$ . First, recurse on  $\ell_0$  using  $\text{sup} \cup r_0$  as the support set. The returned set  $\ell$  is a minimal subset of  $\ell_0$  that is necessary

```

@pre  $p(\text{sup} \cup S_0) \wedge \neg p(\text{sup})$ 
@post  $V \subseteq S_0 \wedge p(\text{sup} \cup V)$ 
       $\wedge \forall e \in V. \neg p(\text{sup} \cup V \setminus \{e\})$ 
let rec min p sup S0 =

```

Fig. 3. Annotated prototype of min, where  $V$  is the return value

for  $p(\text{sup} \cup \ell \cup r_0)$  to hold. Second, recurse on  $r_0$  using  $\text{sup} \cup \ell$  (note:  $\ell$ , not  $\ell_0$ ) as the support set. The returned set  $r$  is a minimal subset of  $r_0$  that is necessary for  $p(\text{sup} \cup \ell \cup r)$  to hold. Finally, return  $\ell \cup r$ , which is a minimal subset of  $S_0$  for  $p(\text{sup} \cup \ell \cup r)$  to hold.

Figure 2 gives a precise definition of this algorithm. To find a minimal subset of  $S$  that satisfies  $p$ , `min` is initially called with an empty support set ( $\square$ ) and  $S$ .

**Theorem 1 (Correct)** *Suppose that  $S$  is nonempty,  $p(S)$  is true, and  $p(\emptyset)$  is false.*

- 1) `min p [] S` terminates.
- 2) Let  $\bar{S} = \text{min } p \ [] \ S$ . Then  $p(\bar{S})$  is true, and for each  $e \in \bar{S}$ ,  $p(\bar{S} \setminus \{e\})$  is false.

*Proof:* The first claim is easy to prove: each level of recursion operates on a finite nonempty set that is smaller than the set in the calling context.

For the second claim, we make an inductive argument of correctness. We prove first that  $p(\bar{S})$  is true. We then prove that for each  $e \in \bar{S}$ ,  $p(\bar{S} \setminus \{e\})$  is false. To prove these claims, we prove that five assertions are inductive for `min`. These assertions are summarized as function preconditions and function postconditions of `min` in Figure 3. Throughout the proof, let  $V = \text{min } p \ \text{sup} \ S_0$  be the return value.

For the first part of the second claim, we establish the following invariants:

- 1)  $p(\text{sup} \cup S_0)$
- 2)  $p(\text{sup} \cup V)$

Invariant (1) is a function precondition of `min`; invariant (2) is a function postcondition of `min`. Hence, the inductive argument for (1) establishes that it always holds upon entry to `min`, while the inductive argument for (2) establishes that it always holds upon return of `min`.

Invariants (1) and (2) are proved simultaneously. For the base case of (1), note that  $p(\emptyset \cup S) = p(S)$ , which is true by assumption. For the inductive case, consider that  $p(\text{sup} \cup \ell_0)$  and  $p(\text{sup} \cup r_0)$  are checked before the first two recursive calls; that  $\text{sup} \cup r_0 \cup \ell_0 = \text{sup} \cup S_0$  for the third recursive call; and that  $p(\text{sup} \cup r_0 \cup \ell)$  is true by invariant (2).

For the base case of invariant (2), we know at the first return of `min` that  $p(\text{sup} \cup S_0)$  from invariant (1), and  $V = S_0$ . For the inductive case, consider that (2) holds at the next two returns by the inductive hypothesis; and that at the fourth return,  $p(\text{sup} \cup \ell \cup r)$  holds by the inductive hypothesis of the prior line.

In the first call to `min` in `minimal`,  $\text{sup} = \emptyset$ ; hence,  $p(\bar{S}) = p(\emptyset \cup \bar{S}) = \text{true}$  by invariant (2).

To prove that  $\bar{S}$  is minimal (that for each  $e \in \bar{S}$ ,  $p(\bar{S} \setminus \{e\})$  is false) for the second part of the second claim, consider the following invariants:

- 3)  $V \subseteq S_0$
- 4)  $\neg p(\text{sup})$
- 5)  $\neg p(\text{sup} \cup V \setminus \{e\})$  for  $e \in V$

Invariant (4) is a function precondition, and invariants (3) and (5) are function postconditions.

For invariant (3), note for the base case that the first return of `min` returns  $V = S_0$  itself; that the next two returns hold by inductive hypothesis; that  $\ell \subseteq \ell_0$  and  $r \subseteq r_0$  by inductive hypothesis; and, thus, that  $V = \ell \cup r \subseteq \ell_0 \cup r_0 = S_0$ .

For the base case of invariant (4), consider that  $\neg p(\emptyset)$  by assumption. For the inductive case, consider that the first two recursive calls have the same  $\text{sup}$  as in the calling context and thus (4) holds by inductive hypothesis; that at the third recursive call,  $\neg p(\text{sup} \cup r_0)$ ; and that at the fourth recursive call,  $\neg p(\text{sup} \cup \ell_0)$  and, from (3), that  $\ell \subseteq \ell_0$ , so that  $\neg p(\text{sup} \cup \ell)$  follows from monotonicity of  $p$ .

For the base case of invariant (5), consider that at the first return,  $\neg p(\text{sup})$  by invariant (4). Hence, the one element of  $S_0$  is necessary. The next two returns hold by the inductive hypothesis. For the final return, we know by the inductive hypothesis that  $\neg p(\text{sup} \cup \ell \cup r \setminus \{e\})$  for  $e \in r$ ; hence, all of  $r$  is necessary. Additionally, from the inductive hypothesis,  $\neg p(\text{sup} \cup r_0 \cup \ell \setminus \{e\})$  for  $e \in \ell$ , and  $\neg p(\text{sup} \cup r_0 \cup \ell \setminus \{e\})$  implies that  $\neg p(\text{sup} \cup r \cup \ell \setminus \{e\})$  by monotonicity of  $p$  and because  $r \subseteq r_0$  by invariant (3); hence, all of  $\ell$  is necessary.

In the first call to `min` at `minimal`,  $\text{sup} = \emptyset$  and  $V = \bar{S}$ ; hence,  $\neg p(\bar{S} \setminus \{e\})$  for  $e \in \bar{S}$  from invariant (5). ■

**Theorem 2 (Upper Bound)** *Let  $\bar{S} = \text{min } p \ [] \ S$ . Discovering  $\bar{S}$  requires making  $O\left((|\bar{S}| - 1) + |\bar{S}| \lg \frac{|S|}{|\bar{S}|}\right)$  queries to  $p$ .*

*Proof:* Suppose that  $|\bar{S}| = 2^k$  and  $|S| = n2^k$  for some  $k, n > 0$ . Each element of  $\bar{S}$  induces one divergence at some level in the recursion. At worst, these divergences occur evenly distributed at the beginning, inducing  $|\bar{S}|$  separate binary searches over sets of size  $\frac{|S|}{|\bar{S}|}$ . Hence,  $|\bar{S}| - 1$  calls to `min` diverge, while  $|\bar{S}| \lg \frac{|S|}{|\bar{S}|}$  calls behave like in a binary search. Noting that each call results in at most two queries to  $p$ , we have the claimed upper bound in this special case, which is also an upper bound for the general case. (Adding sufficient “dummy” elements to construct the special case does not change the asymptotic bound.) ■

For studying the lower bound on the complexity of the problem, suppose that  $S$  has precisely one minimal satisfying subset.

**Theorem 3 (Lower Bound)** *Any algorithm for determining the minimal satisfying subset  $\bar{S}$  of  $S$  must make  $\Omega\left(|\bar{S}| + |\bar{S}| \lg \left(\frac{|S| - |\bar{S}|}{|\bar{S}|}\right)\right)$  queries to  $p$ .*

*Proof:* For the linear component,  $|\bar{S}|$ , consider deciding whether  $\bar{S}$  is indeed minimal. Since all that is known is that  $p$

```

let rec min f sup S0 =
  if |S0| = 1
  then (sup, S0)
  else let ℓ0, r0 = split S0 in
        let v, C = f(sup ∪ ℓ0) in
        if v
        then min f (sup ∩ C) (ℓ0 ∩ C)
        else let v, C = f(sup ∪ r0) in
              if v
              then min f (sup ∩ C) (r0 ∩ C)
              else let C, ℓ = min f (sup ∪ r0) ℓ0 in
                    let sup = sup ∩ C in
                    let r0 = r0 ∩ C in
                    let C, r = min f (sup ∪ ℓ) r0 in
                    (sup ∩ C, (ℓ ∩ C) ∪ r)
let minimal f S =
  let _, S0 = min f [] S in S0

```

Fig. 4. Optimal minimal with additional information

is monotone over  $S$ , the information that  $p(S_0)$  is false does not reveal any information about  $p(S_1)$  when  $S_1 \setminus S_0 \neq \emptyset$ . Therefore,  $p$  must be queried for each of the  $|\bar{S}|$  immediate strict subsets of  $\bar{S}$ .

For the other component, consider that any algorithm must be able to distinguish among  $C(|S|, |\bar{S}|) = \frac{|S|!}{|S|!(|S|-|\bar{S}|)!}$  possible results using only queries to  $p$ . Thus, the height of a decision tree must be at least  $\lg C(|S|, |\bar{S}|)$ . Using Stirling’s approximation,

$$\begin{aligned}
\lg \frac{|S|!}{|S|!(|S|-|\bar{S}|)!} &\geq \lg |S|! - \lg |\bar{S}|! - \lg (|S| - |\bar{S}|)! \\
&\quad - o(\lg |\bar{S}| + \lg (|S| - |\bar{S}|)) \\
&= \Omega \left( |S| \lg \left( \frac{|S|}{|S|-|\bar{S}|} \right) + |\bar{S}| \lg \left( \frac{|S|-|\bar{S}|}{|S|} \right) \right).
\end{aligned}$$

Hence, the algorithm is in some sense optimal. However, a set can have a number of minimal subsets exponential in its size. In this situation, the lower bound analysis does not apply.

In practice, one can often glean more information when executing the predicate  $p$  than just whether it is satisfied by the given set. For example, a decision procedure for propositional satisfiability (a “SAT solver”) can return an *unsatisfiable core*. Hence, if  $\psi \Rightarrow c$  holds ( $\psi \wedge \neg c$  is unsatisfiable), the procedure might return a subclause  $d \sqsubseteq c$  such that  $\psi \Rightarrow d$  also holds. However,  $d$  need not be minimal. The algorithm of Figure 4 incorporates this extra information. Rather than a predicate  $p$ , it accepts a function  $f$  that returns two values:  $f(S)$  returns the same truth value as  $p(S)$ ; and if  $p(S)$  is true, it also returns a subset  $S_0 \sqsubseteq S$  such that  $p(S_0)$  holds. This subset is used to prune sets appropriately. Additionally,  $\min$  returns both the minimal set and a pruned support set to use on the other branch of recursion.

## V. EXPERIMENTS

### A. Implementation

We implemented our analysis in O’Caml. We discuss important elements of our implementation.

1) *SAT Solver*: We instrumented Z-Chaff version “2004.11.15 Simplified” [10] to return original unit clauses that are leaves of the implication graph to aid in computing minimal implicates. We also refined its memory usage to allow tens of thousands of incremental calls. For parallel executions, we tuned Z-chaff to randomize some of its choices.

Conversion to CNF is minimized by caching the CNF version of the transition system within the SAT solver. Also, multiple versions of the transition relation are stored; each version corresponds to a particular slicing of the relation according to the one-step cone of influence.

2) *Depth-First Search*: Our implementation takes a depth-first approach: if it fails to find an inductive clause excluding a CTI, it focuses on this subgoal before again considering the rest of the given property.

3) *Parallel Algorithm*: Each process works mostly independently, relying on the randomness of the SAT solver to focus on different regions of the possible state space of the system. Upon discovery of an inductive clause, a process reports it to a central server and receives all other inductive clauses discovered by other processes since its last report. Because of the depth-first treatment of counterexample states, a process can report that a clause is inductive *under the assumption that subgoal states are unreachable*. If this assumption is incorrect, the process eventually discovers a counterexample trace. Otherwise, it eventually justifies this assumption with additional inductive clauses. However, other processes may finish before receiving these additional clauses. Hence, because only the last process to terminate receives all clauses, it is the only process that is guaranteed to have an inductive strengthening of the safety property.

### B. Benchmarks

1) *PicoJava II Set*: We applied our analysis to the PicoJava II microprocessor benchmark set, previously studied in [11]–[13]. Each benchmark asserts a safety property about the instruction cache unit (ICU) — which manages the instruction cache, prefetches instructions, and partially decodes instructions — but includes the implementation of both the ICU and the instruction folding unit (IFU), which parses the byte stream from the instruction queue into instructions and divides them into groups for execution within a cycle. Including the IFU increases the number of variables in the cone-of-influence (COI) and complicates the combinatorial logic. Hence, for example, a static COI analysis is unhelpful. Of the 20 benchmarks, proof-based abstraction solved 18 [11] (it exhausted the available 512MB of memory on problems PJ<sub>17</sub> and PJ<sub>18</sub>), and interpolation-based model checking solved 19 [12], [13], each within their allotted times of 1000 seconds on 930MHz machines.

2) *VIS Set*: The second set of benchmarks are from the VIS distribution [14]. We applied the analysis to several valid properties of models that are difficult for standard  $k$ -induction (although easy for standard BDD-based model checking) [9].  $k$ -induction with strengthening fails on PETERSON and HEAP

TABLE I  
RESULTS FOR ONE PROCESS

Name	COI	Clauses	SAT queries	Time	Mem (MB)
PJ <sub>2</sub>	306	6 (2)	202 (64)	38s (3s)	212 (9)
PJ <sub>3</sub>	306	6 (3)	201 (78)	37s (3s)	213 (9)
PJ <sub>5</sub>	88	159 (27)	12K (3.4K)	30s (9s)	50 (3)
PJ <sub>6</sub>	318	414 (85)	32K (7.5K)	<b>1h30m (22m)</b>	589 (39)
PJ <sub>7</sub>	67	63 (9)	4K (1K)	10s (2s)	41 (3)
PJ <sub>8</sub>	90	70 (8)	3.5K (.8K)	13s (3s)	43 (3)
PJ <sub>9</sub>	46	27 (5)	1K (.2K)	4s (1s)	35 (2)
PJ <sub>10</sub>	54	6 (3)	213 (110)	6s (1s)	48 (1)
PJ <sub>13</sub>	352	8 (6)	234 (149)	2m45s (1m9s)	379 (15)
PJ <sub>15</sub>	353	145 (68)	6K (3.5K)	30m (17m)	493 (79)
PJ <sub>16</sub>	290	241 (186)	18K (22K)	50m (1h10m)	539 (96)
PJ <sub>17</sub>	211	1.2K (153)	337K (51K)	<b>16h20m (3h)</b>	1250 (110)
PJ <sub>18</sub>	143	740 (152)	91K (23K)	<b>2h40m (50m)</b>	673 (83)
PJ <sub>19</sub>	52	83 (11)	4K (.4K)	11m (5m)	237 (31)
PC <sub>1</sub>	93	7 (4)	170 (105)	2m48s (1m)	360 (12)
PC <sub>2</sub>	91	3 (0)	42 (1)	51s (4s)	335 (1)
PC <sub>5</sub>	91	3 (0)	42 (1)	53s (4s)	335 (1)
PC <sub>6</sub>	91	9 (4)	229 (109)	3m25s (1m18s)	377 (13)
PC <sub>10</sub>	91	21 (10)	598 (260)	5m35s (1m47s)	370 (8)
HEAP	30	2.6K (237)	58K (60K)	<b>4h20m (45m)</b>	330 (25)
PET	16	4 (0)	140 (11)	2s (0s)	44 (0)

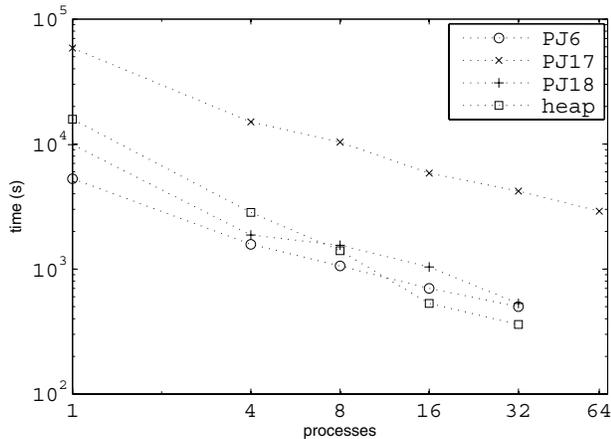


Fig. 5. Time for multiple processes

within 1800 seconds; but BDD-based model checking requires at most a few seconds for each [9].

### C. Results

Table I reports results for executing one process on one processor of a  $4 \times 1.8$ GHz computer with 8GB of available memory. The analysis ran 16 times on each benchmark: Table I reports the number of variables in the cone of influence and the mean and standard deviation (in format *mean (std. dev.)*) for the number of discovered clauses, the number of SAT queries made, the required time, and the peak memory usage. Results are reported only for the nontrivial benchmarks: properties of benchmarks 0, 1, 4, 11, 12, and 14 of the PicoJava II set and benchmarks 3, 4, 7, 8, and 9 of the VIS PPC60x\_2 set are inductive. The PicoJava II benchmarks are labeled PJ<sub>x</sub>; the others are VIS benchmarks. All 20 of the PicoJava II benchmarks were solved; three required more than one hour.

Figure 5 reports results as a log-log plot for analyzing PicoJava II benchmarks 6, 17, and 18 and VIS benchmark HEAP with multiple processes on a cluster of computers with  $4 \times 1.8$ GHz processors and 8GB of memory. Results for one processor are the means from Table I. Times for 32 processes are as follows: PJ<sub>6</sub>, 8m; PJ<sub>17</sub>, 70m; PJ<sub>18</sub>, 9m; and HEAP, 6m. PJ<sub>17</sub> completed in 50m with 60 processes. All benchmarks completed within one hour with some number of processes.

The plot suggests that time decreases roughly linearly with more processes, but only HEAP trades processes for time almost perfectly, possibly because it requires the most clauses. Suboptimal scaling results from generating redundant clauses.

## VI. RELATED WORK

### A. Qualitative Comparisons

We compare the characteristics of several safety analyses: bounded model checking (BMC) [15], interpolation-based model checking (IMC) [12], [13], *k*-induction (*k*I) [7]–[9], [16], [17], predicate abstraction with refinement (CEGAR) [18], [19], and our analysis (FSIS). These analyses are fundamentally based on computing an inductive set that excludes all error states; they consider the property to prove during the computation; and they use a SAT solver as the main computational resource.

We now consider their differences.

1) *Abstraction*: IMC and CEGAR compute successively finer approximations to the transition relation. Each approximation causes a certain set of states to be deemed reachable. When this set includes an error state, IMC increments the *k* associated with its postcondition operator, solving larger BMC problems, while CEGAR learns a separating predicate. In contrast, BMC, *k*I, and FSIS operate on the full transition relation. *k*I strengthens by requiring counterexamples to induction to be ever longer paths. FSIS generalizes from CTIs to inductive clauses to exclude portions of the state space.

2) *Use of SAT Solver*: BMC, IMC, and *k*I pose relatively few but difficult SAT problems in which the transition relation is unrolled many times. CEGAR and FSIS pose many simple SAT problems in which the transition relation is not unrolled.

3) *Intermediate Results*: Each major iteration of IMC and CEGAR produces an inductive set that is informative even when it is not strong enough to prove the property. Each successive iteration of FSIS produces a stronger formula that excludes states that cannot be reached without previously violating the property. Intermediate iterations of BMC and *k*I are not useful, although exceptions include forms of strengthening, which we discuss in greater depth below [7]–[9], [17].

4) *Parallelizable*: Only FSIS is natural to make parallel. The difficulty of subproblems grows with successive iterations in BMC, IMC, and *k*I so that parallelizing across iterations is not useful. Each iteration of CEGAR depends on previously learned predicates. For these analyses, parallelization must be implemented at a lower level, perhaps in the SAT solver.

Differences suggest ways to combine techniques. For example, the key methods of FSIS and *k*I can be combined, and FSIS can serve as the model checker for CEGAR.

## B. Other Related Work

*Blocking clauses* are used in SAT-based unbounded model checking [5]. Their discovery is refined to produce *prime* blocking clauses, requiring at worst as many SAT calls as literals [6]. Our minimal algorithm requires asymptotically fewer SAT calls. A similar algorithm has been proposed in a different context [20], but it handles only sets containing precisely one minimal satisfying subset.

Strengthening based on under-approximating the states that can reach a violating state  $s$  is applied in the context of  $k$ -induction [7]–[9], [17]. Quantifier-elimination [7], ATPG-based computation of the  $n$ -level preimage of  $s$  [8], and SAT-based preimage computation [9] are used to perform the strengthening. Inductive generalization can eliminate exponentially more states than preimage-based approaches.

## VII. DISCUSSION

Let us consider the methods of this paper more generally. The fundamental idea is to generalize from counterexamples to induction (CTIs) to simple inductive invariants. Together, the set of simple inductive invariants strengthens the specification to be inductive. Limiting the form of invariants controls computational costs, while using CTIs focuses the analysis on the safety specification. The structure of the analysis allows a parallel implementation.

Two questions are immediate. What are the CTIs? What is the abstract domain for invariant generation? When the invariant generation is based on propagation, as in this paper, these questions are linked: the abstract domain should be conjunctive *with respect to the CTI* so that the best over-approximation to the CTI in the domain is sufficiently precise.

For example, in FSIS, the CTIs are (partial) states that can lead to violations of the given property; and the domain consists of clauses of system variables. Clauses are conjunctive with respect to states like CTIs that ought to be unreachable. We thus start with  $\Pi$  and conjoin invariant clauses to exclude error states until CTIs no longer exist.

As another example, consider the dual analysis in which the set of reachable states is grown until it is inductive without including any  $\neg\Pi$ -states. Now, the CTIs are (partial) states that are reachable in one step from the currently reachable set; and the abstract domain is cubes, which are conjunctions of literals. Hence, the invariant cubes are combined through disjunction to grow the reachable set. Each round of invariant generation discovers a *minimal inductive subcube* of the cube defined by the CTI that includes only  $\Pi$ -states.

In another application, we explored inductive generalization from CTIs to affine inequalities [21]. In the domain of the analysis, invariant generation is not based on propagation.

Once the form of the CTI and the abstract domain are fixed, one desires to find a greatest inductive generalization to each CTI. Standard techniques suggest how to perform one direction of propagation in the abstract domain [2]. However, the other direction must suffer from the nondeterminism inherent in over- (under-) approximating in a disjunctive (conjunctive) domain, so that a greatest inductive generalization

to the CTI need not be unique. For example, *implicate* is nondeterministic; and in the dual analysis, computing a best *implicant* is nondeterministic. The function *minimal* of Section IV-B is a general operator for performing forward (backward) propagation in disjunctive (conjunctive) domains.

This general perspective on the ideas of this paper suggests further work in the form of exploring other domains. Additionally, we intend to combine the method with  $k$ -induction. Finally, our analysis is motivated by a classically deductive approach to verification [1]. We are exploring analyses for other classes of temporal properties that are also motivated by classically deductive techniques.

## ACKNOWLEDGMENTS

The authors wish to thank Prof. A. Aiken, A. M. Bradley, Prof. E. Clarke, Prof. D. Dill, Dr. A. Gupta, Dr. H. Sipma, Prof. F. Somenzi, and the anonymous reviewers for their comments; and Prof. A. Aiken for the use of his computer cluster.

## REFERENCES

- [1] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. New York: Springer-Verlag, 1995.
- [2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM Press, 1977, pp. 238–252.
- [3] B. Knaster, "Un theoreme sur les fonctions d'ensembles," *Ann. Soc. Polon. Math.*, vol. 6, pp. 133–134, 1928.
- [4] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications," *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955.
- [5] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 250–264.
- [6] H. Jin and F. Somenzi, "Prime clauses for fast enumeration of satisfying assignments to boolean circuits," in *DAC*. ACM Press, 2005.
- [7] L. de Moura, H. Ruess, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *CAV*, ser. LNCS. Springer, 2003.
- [8] V. C. Vimjam and M. S. Hsiao, "Fast illegal state identification for improving SAT-based induction," in *DAC*. ACM Press, 2006.
- [9] M. Awedh and F. Somenzi, "Automatic invariant strengthening to prove properties in bounded model checking," in *DAC*. ACM Press, 2006.
- [10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *DAC*, 2001.
- [11] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *TACAS*, ser. LNCS, vol. 2619. Springer, 2003, pp. 2–17.
- [12] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 1–13.
- [13] —, "Applications of Craig interpolants in model checking," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 1–12.
- [14] *VIS*. [Online]. Available: <http://visi.colorado.edu/~vis>
- [15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*. Springer, 1999, pp. 193–207.
- [16] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, ser. LNCS, vol. 1954. Springer, 2000.
- [17] R. Armoni, L. Fix, R. Fraer, S. Huddleston, N. Piterman, and M. Vardi, "SAT-based induction for temporal safety properties," in *BMC*, 2004.
- [18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.
- [19] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word level predicate abstraction and refinement for verifying RTL verilog," in *DAC*, 2005.
- [20] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC / SIGSOFT FSE*, 1999, pp. 253–267.
- [21] A. R. Bradley and Z. Manna, "Verification constraint problems with strengthening," in *ICTAC*, ser. LNCS, vol. 3722. Springer, 2006.

# Fast Minimum-Register Retiming via Binary Maximum-Flow

Aaron P. Hurst, Alan Mishchenko, and Robert K. Brayton  
University of California, Berkeley, CA

**Abstract**— We present a formulation of retiming to minimize the number of registers in a design by iterating a maximum network flow problem. The retiming returned will be the optimum one, which involves the minimum amount of register movement. Existing methods solve this problem as an instance of minimum-cost network flow, an asymptotically and practically more difficult problem than maximum flow. Furthermore, because all flows are unitary, the problem can be simplified to binary marking. Our algorithm has a worst-case bound of  $O(R^2E)$ , where  $R$  is the number of registers and  $E$  the number of pair-wise connections. We demonstrate on a set of circuits that our formulation is 5x faster than minimum-cost-based methods.

**Index Terms**—Retiming, Sequential Verification, State Minimization, Maximum Flow.

## I. INTRODUCTION

RETIMING [13] moves registers over combinational nodes in a logic network, preserving output functionality and logic structure. Retiming can target a number of objectives: (i) minimize the delay of the circuit (*min-delay*), (ii) minimize the number of registers under a delay constraint (*constrained min-register*), or (iii) minimize the number of registers (*unconstrained min-register*). Numerous approaches have been proposed to achieve these goals [13]-[18], with most of the emphasis on the first two objectives.

In this paper, we focus on unconstrained min-register retiming, which has several applications in logic synthesis and verification. In synthesis, minimizing the number of registers can save area and power or be used to ameliorate local congestion. Even if delay constraints are ignored, often any timing violations can be corrected with logic sizing, combinational resynthesis, or intentional clock skewing [8]. In verification, min-register retiming minimizes the number of state variables [12], which reduces the size of the sequential verification problem and has been demonstrated to reduce the overall difficulty [3]. This may be critical for proof completion.

Although retiming problems are traditionally expressed as general linear programs, they can be solved efficiently as minimum-cost network circulation problems using suitable algorithms. Instead, we propose a retiming method that is based on iterated binary maximum network flow, which is a computationally easier problem to solve. The number of iterations required appears to be quite small in practice. Because the result of each iteration is strictly better than the

previous one, the computation can be bounded and still result in an improvement. It was found experimentally that the first iteration accounts on average for 90% of the total reduction in the number of registers. This can be used to trade quality for runtime when a problem is large or when fast computation is critical.

To support these claims, we provide experimental results on moderately-sized industrial benchmarks and a few larger artificial ones. They demonstrate the efficiency of the new algorithm: the optimum result can be generated for circuits with more than a million gates in less than a minute and much faster than using existing methods. On pre-optimized benchmark circuits, the average reduction in register count was 11%, ranging between 0% and 63%.

An important feature of our algorithm is that it always returns the minimum-register retiming that is closest to the current position of the registers. If a register in the input circuit cannot be retimed to minimize the total register count, it is not touched. This simplifies the computation of the initial states and minimizes the total perturbation.

The paper is organized as follows. Section II describes the background information and existing approaches to minimum-area retiming. Section III describes the new algorithm. Section IV reports experimental results.

## II. BACKGROUND

A circuit is modeled as a directed graph  $\mathcal{G}=\langle V,E \rangle$  whose vertices  $V$  correspond to logic gates and directed edges  $E$  correspond to wires connecting the gates, decomposed into pair-wise connections from gate outputs to inputs. The circuit's external connections are represented by additional primary input (PI) and primary output (PO) vertices. The terms *network*, *graph*, and *circuit* are used interchangeably in this paper.

A node has zero or more fan-ins, i.e. nodes that are driving this node, and zero or more fan-outs, i.e. nodes driven by this node. The transitive fan-out of a vertex  $v$  is a subset of all nodes of the network reachable through the fan-out edges from  $v$ , captured by the function  $\text{TFO}(v): V \rightarrow 2^V$ .

A *combinational frame* of the circuit is comprised of the acyclic network between the register outputs / PIs and register inputs / POs. An example of this is illustrated in Figure 1; the inputs (the register outputs) lie on the left, and the outputs (the register inputs) on the right. The registers are duplicated for the ease of illustration.

### A. Retiming

The problem of retiming is to relocate the registers in a circuit to optimize some circuit characteristic while preserving output functionality (and optionally meeting some additional constraints). The repositioning is captured by a *retiming lag function*  $r(v):V \rightarrow \mathbb{Z}$  that describes the number of registers moved backward over each combinational node. There are several formulations of the retiming problem, but for the purposes of this paper, we utilize the linear program (LP) in Equations 1-2 from [13].  $w_i(e)$  is the initial number of registers present on edge  $e$ .

$$\min \sum_{\forall e=(u,v)} r(v) - r(u) \text{ s.t.} \quad (1)$$

$$r(u) - r(v) \leq w_i(e), \quad \forall e = (u,v) \quad (2)$$

The dual of this LP is a minimum-cost network circulation problem and can be solved efficiently using specialized algorithms. Using the method described by Goldberg [9], the minimum-cost flow can be computed in  $O(VE \log(V^2/E) \log(VC))$  worst-case time, where  $C$  is the maximum cost on any edge. The number of vertices and edges in the corresponding network problem is proportional to the size of the combinational circuit.

After a retiming has been determined, an equivalent set of initial values must be computed for the registers. If no such initial state exists, the particular retiming must be rejected or the circuit otherwise altered. We discuss consequences for this computation but refer to [19] for its details.

### III. MIN-REG RETIMING ALGORITHM

We introduce an algorithm for optimum unconstrained minimum-register retiming that is based on an iterative maximum network flow problem. This is motivated by the observation that computing the maximum flow through a network is an asymptotically and practically easier problem than determining the minimum-cost circulation. Our algorithm requires repeated iteration, but for practical circuits, the number of iterations is typically quite small.

#### A. Single Iteration

The core of the algorithm consists of minimizing the number of registers within a single combinational frame. Let us consider only the paths through the combinational logic that lie between two registers (thus temporarily ignoring the primary inputs and outputs). The current position of the registers clearly forms a complete cut through the network, immediately at its inputs. The width of the cut is the initial number of registers.

Consider retiming the registers in the forward direction through the circuit. As the registers are retimed over the combinational nodes, the corresponding cut moves forward through the network and may grow or shrink in width as registers are replicated and/or shared as dictated by the graph structure. In the initial circuit, it is evident that any path in the combined graph passes through exactly one register, and any

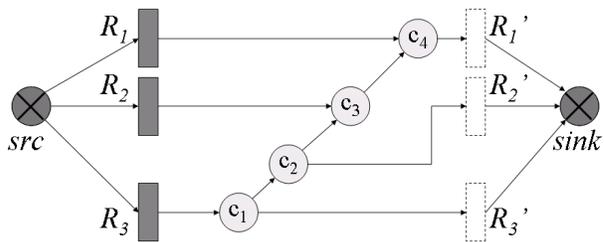


Fig. 1. An example of the flow problem for a small circuit. The combinational elements are light grey; the initial positions of the registers lie to the left, and their inputs are replicated on the right. This is also an example of a network where the minimum-cut in the directed graph is not a valid retiming: the graph can be completely cut with exactly two registers (at the outputs of  $c_1$  and  $c_4$ ), but this results in a path  $(R_3 \rightarrow R_1')$  with altered sequential latency.

valid retiming must preserve this property. If this were not the case, the latency of that path would be altered and the sequential behavior of the circuit changed.

The problem of minimizing the number of registers by retiming them to new positions within the scope of the combinational frame is equivalent to finding a minimum width cut. This is the dual of the maximum network flow problem, for which efficient solutions exist. To construct the flow problem, the flow source is connected to all register outputs, and the register inputs are connected to the flow sink. For now, the capacity of all edges is assumed to be one.

Solving the general maximum flow problem using the preflow-based method proposed by Cherkassky and Goldberg [4] is  $O(VE \log(V^2/E))$  in the worst case. Alternatively, the

classical augmenting-path-based methods (e.g. [6]) are bounded by the maximum length of an augmenting path times the maximum flow,  $U$ :  $O(UE)$ . Because the flow constraints in our problem are of unit capacity, the width of the input to the graph establishes a worst-case bound of  $O(RE)$ , where  $R$  is the initial number of registers. The worst-case bound on the maximum flow problem using either the preflow- or augmenting-path-based algorithm is therefore asymptotically faster than the best known bound on the minimum-cost flow problem on an identical graph structure.

Once the maximum flow has been established, we have available the residual graph that describes the remaining edge capacities with respect to this flow. The residual graph is used to generate a corresponding minimum cut. Via duality, the width of this minimum cut is exactly the volume of the maximum flow and the flow through the saturated edges crossing the cut. To determine its location, the vertices in the network are partitioned into two sets:  $S$ , those that are reachable in the residual graph with additional flow from the source, and  $R$ , those that are not. Generating this partition is  $O(E)$  in the worst case. The partition must be complete cut, because there can exist no additional flow path from the source to the sink if the maximum has already been assigned. For every edge  $u \rightarrow v$  such that  $(u \in S) \wedge (v \in R)$ , a register must be placed on the output of the gate associated with  $u$ . The registers are removed from their current locations and placed

on the graphs edges that cross the minimum cut.

There may exist multiple cuts of minimum width, but this method always generates the one that is unambiguously closest to the source node. This results in the minimal movement of the registers, simplifying the initial state computation and minimizing the design perturbation.

However, as stated, this procedure may generate an illegal retiming. A cut in a directed graph only guarantees that all paths in the graph are cut *at least* once. This is a necessary but not sufficient condition for the cut to be a valid retiming. We seek the minimum cut in the graph such that all paths are crossed *exactly* once. Figure 1 illustrates an example of this problem. The network flow problem must be altered to eliminate the possibility that a path is crossed more than once.

*Reverse edges* with unbounded capacity are added in the direction opposite to the constrained edges in the original network. These additional paths may increase the maximum flow (and therefore the size of the minimum cut) but guarantee that the resulting minimum cut will correspond to a legal retiming. For a path in the original graph to cross the finite-width cut more than once from  $S \rightarrow R$ , there must be at least one edge that crosses from  $R \rightarrow S$ . If the unbounded reverse edges are also present, this would imply an infinite-capacity edge from  $S \rightarrow R$ , thus violating the finite cut-width assumption and ensuring that such a cut will not be generated.

It is also needed to account for the sharing of registers at nodes with multiple fan-outs. This requires another simple modification to the network flow problem. Each circuit node is decomposed into two vertices: a *receiver* of all of the former fan-in arcs and an *emitter* of all of the former fan-out arcs. The flow constraints are removed from these edges, and a single edge with a unit flow constraint is inserted from the receiver to the emitter. Then, to model fan-out (as opposed to fan-in) sharing, the reverse edges are connected between adjacent receivers. As described above, the unconstrained edges can not participate in the minimum cut; only the internal edge is available to make a unit contribution to the cut-width. Each node will therefore require at most one register regardless of its fan-out degree. This idea can also be extended to model fan-in sharing as in [1].

The final network for computing the maximum flow computation is depicted in Figure 2.

The unitary flow constraints can also be used to simplify the implementation of the maximum flow solver such that the flow network of Figure 2 need not be explicitly built. Both the preflow- and augmenting-path-based techniques can be performed directly on the original circuit with binary marking. Because the flows on the non-unit arcs are unconstrained, they need only be implicitly maintained with predecessor pointers.

In summary, a single iteration of our retiming algorithm involves computing the maximum flow through the combinational frame, identifying the unique topologically closest minimum cut, and moving the register boundary to the new location.

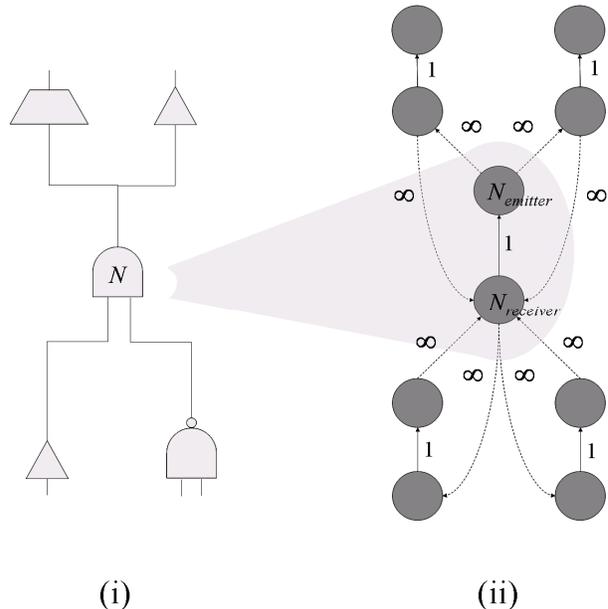


Fig. 2. The circuit hypergraph around node  $N$  in (i) is expanded to form the digraph network flow problem (ii) to compute a valid forward retiming with maximal fan-out sharing. Each of the gates in (i) is replaced with a pair of dark grey nodes in (ii) and the flow arcs as illustrated. The capacity of each arc is labeled (with solid edges having unit capacity and dashed edges being unconstrained).

### B. Primary Inputs and Outputs

The primary inputs and outputs (PIOs) can be treated in different ways, depending on the application. The allowed locations of the minimum cut and the subsequent insertion/removal of registers can be adjusted to either fix or selectively alter the sequential behavior of the circuit.

In synthesis, the relative latencies at all of the PIOs is assumed to be invariant. In verification applications, it is not necessary to preserve the synchronization of the inputs and outputs. It may be desirable to borrow or loan registers to the environment individually for each PIO if the result is a net decrease in the total register count.

To allow register borrowing, the external connections should be left dangling. Registers will be donated to the environment if the minimum cut extends past the dangling primary outputs (POs); conversely, registers will be borrowed if the minimum cut appears in the transitive fan-out region of the dangling primary inputs (PIs). The inclusion of this region introduces additional flow paths and allows additional possibilities for minimizing the total register count.

To disallow desynchronization with the environment, a host node and normalization can be employed, or alternatively, the flow problem suitably modified: the POs are connected to the sink and the transitive fan-out of the PIs is blocked from participating in the minimum cut. All paths through the combinational network that originate from a PI have a sequential latency that must remain at zero; inserting a register anywhere in the  $TFO(\{PIs\})$  would alter this. To exclude this

region, one of two methods can be used: (i) temporarily redirecting to the sink all edges  $e=(u,v)$  where  $v \in \text{TFO}(\{\text{PIs}\})$ , or (ii) replacing the constrained flow arcs in this fan-out cone with unconstrained ones, thus preventing these nodes from participating in the minimum cut. Both methods restrict the insertion or deletion of registers in the invalid region.

Disallowing desynchronization during verification may be motivated by the need to control complexity. Because register borrowing requires the initial values of the new registers to be constrained to those reachable in the original circuit, it is necessary to construct additional combinational logic for computing the initial state. If the size of this logic grows undesirably large, register borrowing can be turned off at any point for any individual inputs or outputs.

### C. Multiple Iterations

This section describes how to compute the globally optimum min-register retiming by iteratively applying the maximum-flow algorithm of Section III.B.

Thus far, we have only considered the forward retiming of registers in the circuit. It is sufficient to consider only one direction if the circuit is strongly connected (i.e. through the use of a host node and normalization). However, in general, the optimum minimum-register retiming requires both forward and backward moves. The procedure for a single iteration of backward retiming is essentially identical, except that it computes the maximum flow from the register inputs (sources) to the primary inputs and register outputs (sinks).

The overall algorithm consists of two iterative phases: forward and backward. The procedure is outlined in Algorithm 1. In each phase, the single-frame optimization is repeated until the number of registers reaches a fix-point. In terms of the retiming lag function, each node's lag is either unchanged or changed by one in each iteration.

At no point during retiming is it necessary to unroll the circuit or alter the combinational logic; only the registers are moved by extracting them from their initial positions and inserting them in their final ones. This modification does change the definition of the combinational frame and the connections to the flow source and sink.

ALGORITHM 1: UNCONSTRAINED MIN-REGISTER RETIMING

---

```

1 while(improvement) { // forward
2   init forward retiming flow network
3   mark PI restricted locations
4   compute maximum flow
5   derive nearest minimum cut
6   move registers to cut
7 }
8 while(improvement) { // backward
9   init backward retiming flow network
10  mark PO restricted locations
11  compute maximum flow
12  derive nearest minimum cut
13  move registers to cut
14 }
15 compute initial states

```

---

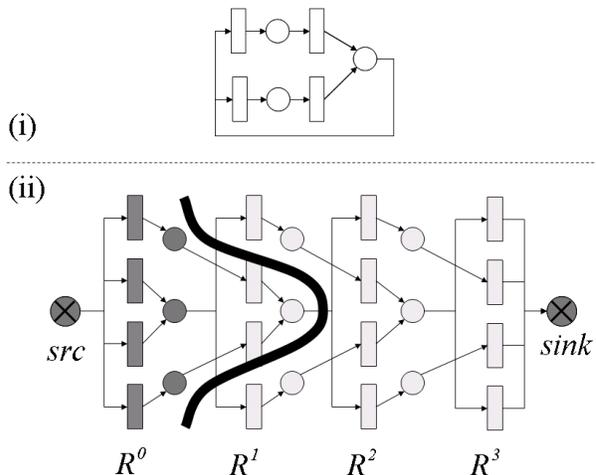


Fig. 3. The circuit in (i) is unrolled three times into (ii). Each copy comprises a combinational frame, the first of which is dark gray. The inter-frame register boundaries are labeled with the corresponding frame number. The globally minimum width retiming cut of the registers in the reference frame  $R^0$  is illustrated by the bold line. Because the minimum width retiming cut stretches across multiple frames, its computation would therefore require multiple iterations of our algorithm.

The ordering of the two phases (forward and backward) doesn't affect the number of registers in the result, but we chose to perform forward retiming first because the path of moves towards the min-register retiming is in general not unique. This approach reduces the amount of logic that has to be retimed in the backward direction, thereby reducing the difficulty of computing a new initial state [19].

### D. Proof of Correctness

Given a normalized retiming lag function  $r(v): V \rightarrow \mathbb{Z}^{0,+}$ , consider unrolling the sequential circuit by  $n$  cycles, where  $n > \max_{\forall v} r(v)$ . This corresponds to stacking multiple combination frames, as illustrated in Figure 3.

The positions of the registers of the reference cycle after any retiming  $r(v)$  can be expressed as a cut  $C$  in the edges of this unrolled circuit. The elements of  $C$  are the retimed register positions. The unretimed cut,  $C_{init}$ , (such that  $r(v)=0$ ) lies at the base of the unrolled circuit. The size of this cut,  $|C|$ , is the number registers post-retiming, or equivalently, the number of combinational nodes with some fan-out crossing the cut.

A cut  $C$  is a *valid retiming* if every path through the combinational network passes through it exactly once. This implies for any two registers  $R_1, R_2 \in C$  that  $R_1 \cap \text{TFO}(R_2) = \emptyset$  and vice versa. If this were not the case, additional latency would be introduced and the functionality of the circuit would be altered.

Consider an optimal minimum register retiming and its corresponding cut  $C_{min}$ . While there exist many such cuts, assume  $C_{min}$  to be the one that lies strictly forward of the initial register positions is topologically closest to  $C_{init}$ . It can be shown with Lemma 1 that there is one unambiguously closest cut.

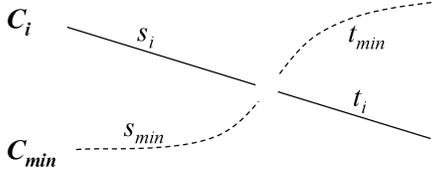


Fig. 4. Definitions of cut partitions for Section III.D.

**Theorem 1:** Upon termination of our algorithm, the resulting cut is exactly  $C_{min}$ .

*Proof.* Our algorithm iteratively computes the nearest cut of minimum width reachable within one combinational frame and terminates when there is no change in the result. Let the resulting cut after iteration  $i$  be  $C_i$ . The cut  $C_i$  at termination will be identical to  $C_{min}$  if the following two conditions are met.

*Condition 1.* No register in  $C_i$  lies topologically forward of any register in  $C_{min}$ .

*Condition 2.* After each iteration,  $|C_{i+1}| < |C_i|$  unless  $C_i = C_{min}$ .

*Lemma 1.* Let  $C_i$  and  $C_j$  be two valid retiming cuts, and  $\{s_i, t_i\}$  and  $\{s_j, t_j\}$  be a partitioning of each:  $(s \cup t = C) \wedge (s \cap t = \emptyset)$ . Also for any path  $p$ ,  $(p \cap s_i \Leftrightarrow p \cap s_j)$  and  $(p \cap t_i \Leftrightarrow p \cap t_j)$ . If this is the case, the cuts  $\{s_i, t_j\}$  and  $\{s_j, t_i\}$  are also valid retimings.

One example of such a partitioning is induced by topological order. If the points of intersection of the cuts with a path  $p$  are  $R_i \in C_i$  and  $R_j \in C_j$ , we can assign the registers to  $s$  if  $R_j \in \text{TFO}(R_i)$ , and  $t$  otherwise. The  $s$  sets will include the registers that are topologically closer in  $C_i$  than  $C_j$ ; the  $t$  sets will include the registers that are in both sets or topologically closer in  $C_j$  than  $C_i$ .

If a given  $p$  crosses  $C_i$  at  $R_i \in s_i$ , it may not cross any other register in  $t_i$  (from the definition of a partition). It also may not contain any register in  $t_j$  (from the definition of the sets). The cut  $\{s_i, t_j\}$  has no more than one register on any path.

If a  $p$  does *not* intersect  $s_i$ , then we know that it must cross at some  $R_i \in t_i$  (from the definition of a partition). It also must then intersect some register in  $t_j$  (from the definition of the sets). The cut  $\{s_i, t_j\}$  has at least one register on any path.

Therefore,  $\{s_i, t_j\}$  is crossed by every path exactly once and is a valid retiming. Similarly for  $\{s_j, t_i\}$ .

*Proof of Condition 1.* Consider a cut  $C_i$  that violates Condition 1. Let  $\{s_i, t_i\}$  be a partition of  $C_i$  and  $\{s_{min}, t_{min}\}$  be a partition of  $C_{min}$  such that  $s_i$  is the subset of registers in  $C_i$  that lie topologically forward of the subset  $s_{min}$  of the registers in  $C_{min}$ . This is illustrated in Figure 4. By Lemma 1, we know that both  $\{s_i, t_{min}\}$  and  $\{s_{min}, t_i\}$  are valid cuts.

Because a single iteration returns the closest cut of minimum width within a frame, this  $C_i = \{s_i, t_i\}$  must be strictly smaller than the closer  $\{s_{min}, t_i\}$ . This implies that  $|s_i| < |s_{min}|$  and that  $|\{s_i, t_{min}\}| < |\{s_{min}, t_{min}\}| = |C_{min}|$ . This contradicts the assumption that  $C_{min}$  is optimal, and condition 1 must be true.

*Observation 1.* Retiming by an entire combinational frame does not change any of the register positions in the resulting circuit and also represents a valid retiming cut. Because a register is moved over every combinational node, the retiming lag function is universally incremented. The number of registers on a particular edge is a relative quantity, and the result is structurally identical to the original.

*Proof of Condition 2.* We can use the minimum cut to generate a cut that is strictly smaller than  $C_i$  and reachable within a combinational frame. Consider the cut  $C_{min}'$  that is generated from  $C_{min}$  via Observation 1 such that its deepest point is reachable within the combinational frame of  $C_i$ . Some of the retiming lags may be negative. Let  $\{s_i, t_i\}$  be a partition of  $C_i$  and  $\{s_{min}, t_{min}\}$  be a partition of  $C_{min}'$  such that  $s_{min}$  are the deepest registers in  $C_{min}'$  that lie topologically forward of the subset  $s_i$  of the registers in  $C_i$ .  $s_{min} \neq \emptyset$  if  $C_i \neq C_{min}$ . Via Lemma 1, both  $\{s_i, t_{min}\}$  and  $\{s_{min}, t_i\}$  are valid cuts.

We know that  $|s_{min}| < |s_i|$ , otherwise there would be implied the existence of a topologically closer cut  $|\{s_i, t_{min}\}| \leq |C_{min}'|$ . Therefore, the cut  $\{s_{min}, t_i\}$  is strictly smaller than  $C_i$  and is reachable within one combinational frame and would be returned by a single iteration of the algorithm. Note that this doesn't imply that there aren't other smaller cuts, only that there must exist at least one that is strictly smaller. Therefore, Condition 2 must also be true. ■

### E. Complexity Analysis

As described in Section III.A, the complexity of computing the minimum cut in each iteration of our algorithm is  $O(RE)$ . The maximum number of iterations can also be bounded by  $R$  via Condition 2 in the above proof. The total worst-case runtime is therefore  $O(R^2E)$ . While this is strictly non-comparable to the best known bound for the equivalent minimum-cost network flow problem [9], the results in Section IV indicate that the average runtimes are smaller for the considered circuits.

For the set of benchmarks that we examined, the number of iterations required was small: the average was 2.7 with a maximum of 15. Figure 5 illustrates the fraction of the total

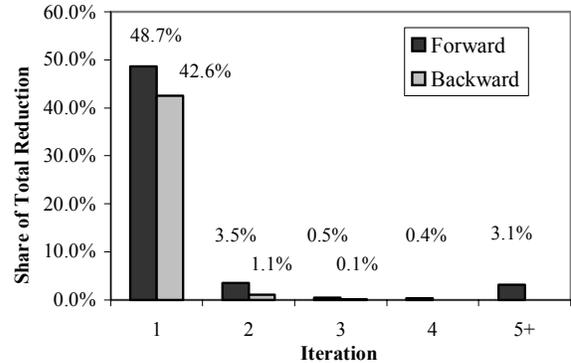


Fig. 5. The contribution of each iteration to the total reduction in the number of registers.

register reduction that was contributed by each iteration. Almost all of the reduction in the number of registers occurs after the first iteration in either direction. The number of iterations can be bounded to a small constant to fix the total worst-case runtime to  $O(RE)$ .

#### IV. EXPERIMENTAL RESULTS

We applied the proposed algorithm to a suite of gate-level circuits derived from public-domain hardware designs [11]. Altera tools were used to extract and optimize the logic networks. This optimization may have already included

reductions in the number of registers. These were then minimally preprocessed by the ABC logic synthesis package [2] as follows: the original hierarchical designs were (a) flattened, (b) structurally hashed and (c) algebraically balanced. From the set of 63 benchmarks, we removed one combinational circuit and 19 circuits whose initial register count was already minimum, leaving 43 circuits shown in Table I.

Our algorithm was implemented in C++. The maximum network flow problem was internally solved using the HIPR package available at [10] and described in [4].

TABLE I: MINIMUM REGISTER RETIMING RESULTS ON REAL BENCHMARKS

Name	Original Circuit			Min-Delay Retiming			Min-Register Retiming				
	AIG	A	D	A	D	T	F-iter	B-iter	A	D	T
barrel16a	397	37	11	124	4	0.02	1	0	32	11	<0.01
barrel16	357	37	10	85	4	0.01	1	0	32	11	<0.01
barrel32	902	70	12	166	5	0.03	1	0	64	13	<0.01
barrel64	2333	135	14	422	5	0.06	1	0	128	14	<0.01
mux32_16bit	1851	533	9	873	4	0.05	1	1	505	11	0.01
mux64_16bit	3743	1046	13	1460	5	0.12	1	0	991	13	0.01
mux8_128bit	3717	1155	7	2297	3	0.18	1	1	1029	8	<0.01
mux8_64bit	1861	579	7	1145	3	0.07	1	1	517	8	<0.01
nut_000	1262	326	58	393	27	0.05	1	2	312	60	<0.01
nut_001	3179	484	93	558	57	0.08	2	2	435	109	0.03
nut_002	873	212	24	232	10	0.02	2	2	158	25	<0.01
nut_003	1861	265	37	304	24	0.04	3	1	228	46	0.01
nut_004	713	185	13	213	6	0.02	2	2	164	15	<0.01
oc_aes_core_inv	11177	669	25	669	25	0.25	1	1	658	25	0.04
oc_aes_core	8732	402	24	402	24	0.14	1	1	394	24	<0.01
oc_aquarius	23109	1477	207	1575	200	0.81	1	0	1473	206	0.08
oc_ata_ocidec1	1601	269	14	275	11	0.02	1	0	268	14	<0.01
oc_ata_ocidec2	1813	303	14	310	11	0.02	1	1	293	14	<0.01
oc_ata_ocidec3	3957	594	14	599	13	0.06	1	1	562	19	<0.01
oc_ata_vhd_3	3933	594	14	599	13	0.06	1	1	568	14	<0.01
oc_ata_v	838	157	14	169	10	0.02	1	0	156	14	<0.01
oc_cfft_1024x12	9498	1051	61	1672	26	0.91	12	1	704	346	0.70
oc_cordic_p2r	8430	719	55	975	45	0.26	1	0	718	55	0.01
oc_dct_slow	879	178	32	207	14	0.03	0	1	176	32	<0.01
oc_des_perf_opt	21281	1976	15	4656	14	1.27	15	0	1015	233	1.18
oc_fpu	16115	659	2661	1578	543	30.65	2	0	247	2712	0.12
oc_hdlc	2221	426	14	426	13	0.03	1	3	375	17	<0.01
oc_minirisc	1918	289	36	290	33	0.03	2	1	253	39	0.01
oc_oc8051	10315	754	92	757	87	0.19	1	1	743	92	0.01
oc_pci	10426	1354	46	1405	26	0.39	1	1	1308	46	0.02
oc_rtc	1093	114	41	114	29	0.02	1	0	86	41	<0.01
oc_sdram	860	112	13	109	12	0.02	1	0	109	12	<0.01
oc_simple_fm_rec	2300	226	66	276	40	0.05	0	1	223	75	<0.01
oc_vga_lcd	9086	1108	35	1126	25	0.24	2	1	1078	35	0.02
oc_video_dct	36465	3549	60	8525	16	12.84	1	1	2305	73	0.30
oc_video_huff_dec	1591	61	21	65	18	0.02	0	1	60	22	<0.01
oc_video_huff_enc	1720	59	19	90	13	0.02	1	0	47	32	<0.01
oc_wb_dma	15026	1775	19	1794	17	0.45	1	1	1751	34	0.08
os_blowfish	9806	891	79	906	61	0.30	1	0	827	78	<0.01
os_sdram16	1156	147	23	162	17	0.02	1	0	144	23	<0.01
radar12	38058	3875	110	3991	56	3.71	2	3	3754	110	0.21
radar20	75149	6001	110	6363	56	6.92	2	1	5364	110	1.34
uoft_raytracer	145960	13079	237	16974	208	23.70	3	2	11610	537	3.76
<b>AVERAGE</b>		<b>1.0</b>	<b>1.0</b>	<b>1.41</b>	<b>0.66</b>				<b>0.89</b>	<b>1.56</b>	

TABLE II: MIN-COST VERSUS ITERATIVE MAX-FLOW MINIMUM REGISTER RETIMING ON LARGE ARTIFICIAL BENCHMARKS

Name	Original circuit		Minimum-Register Retiming					
	AIG	A	A	Min-Cost Flow	Iterative Max-Flow			Speedup
				T	F-iter	B-Iter	R	
large1	1 006 k	72.9 k	66.9 k	147.9s	3	3	33.0s	4.48
large2	1 005 k	82.7 k	76.9 k	131.3s	3	3	24.5s	5.36
deep3	1 010 k	74.7 k	67.6 k	182.0s	3	21	34.2s	5.32
deep4	1 074 k	86.4 k	82.0 k	130.3s	3	3	17.9s	7.27
larger5	2 003 k	151.1 k	139.5 k	410.6s	3	3	67.2s	6.11
largest6	4 008 k	300.1 k	279.0 k	818.3s	3	3	139.9s	5.85

Table I is divided into three groups of columns, each describing the characteristics of a particular retiming. The first section of Table I shows the statistics about the circuit with the registers in their initial positions. The second section describes the results of an incremental heuristic min-delay retiming algorithm [18] implemented in ABC to provide perspective on the area/delay tradeoffs. The third set of columns shows the results produced by the proposed min-register retiming algorithm.

The following notation is used in the table. Columns labeled “A” refer to the number of registers in the network (area). Columns labeled “D” refer to the number of nodes on the longest combinational path. Columns labeled “T” refer to the cumulative runtime of the flow computations in seconds measured on 2.0Ghz Pentium Xeon. For the minimum register retiming algorithms, the number of forward and backward iterations that are required before the fix-point is reached are also listed (“F-iter” and “B-iter”, respectively).

Because these benchmarks are only of moderate size, a set of larger artificial circuits was created by combining the benchmarks in Table I. These are described in Table II. As the number of retiming iterations required appears to be independent of the circuit size—probably because the maximum latency around any loop or from input to output is fairly size independent—the circuits “large1” and “large2” were constructed via parallel composition to preserve this property. The 2 and 4 million gate circuits, “larger5” and “larger6”, were generated similarly. In contrast, the two circuits “deep3” and “deep4” were built by random division and serial composition.

In Table II, the results of our iterative maximum-flow-based algorithm are compared against a single minimum-cost-flow-based implementation as described by [4]. The latest CS2 package from [10] was used as the solver. In every case, the iterative maximum-flow-based implementation required less time to complete; on average, it was 5x faster.

## V. CONCLUSIONS

This paper presented an application of a simplified maximum flow computation to the problem of minimizing the number of registers after retiming. The presented method is very simple, straight-forward to implement, fast, memory efficient, and scalable for large industrial circuits. Potential applications of the method include sequential synthesis and verification.

## ACKNOWLEDGEMENTS

This work was supported by SRC contracts 1361.001 and 1444.001, and the California Micro Program with our industrial sponsors Altera, Intel, Magma, and Synplicity.

## REFERENCES

- [1] J. Baumgartner and A. Kuehlmann, “Min-area retiming on flexible circuit structures”, *Proc. ICCAD '01*, pp. 176-182.
- [2] Berkeley Logic Synthesis and Verification Group, *ABC: A System for Sequential Synthesis and Verification*, Release 61104. <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [3] G. Cabodi, S. Quer and F. Somenzi, “Optimizing sequential verification by retiming transformations”, *Proc. DAC '01*, pp. 601-606.
- [4] B. V. Cherkassky and A. Goldberg, “On Implementing Push-Relabel Method for the Maximum Flow Problem,” *Algorithmica* 19, 1997, pp. 390-410.
- [5] J. Cong and C. Wu, “Optimal FPGA mapping and retiming with efficient initial state computation”, *IEEE Trans. CAD*, vol. 18(11), Nov. 1999, pp. 1595-1607.
- [6] J. Edmonds and R. Karp, “Theoretical improvements in algorithmic efficiency for network flow problems”, *Journal of the ACM*, vol. 19 (2), 1972, pp. 248-264.
- [7] G. Even, I. Y. Spillinger, and L. Stok, “Retiming revisited and reversed”, *IEEE Trans. CAD*, vol. 15(3), March 1996, pp. 348-357.
- [8] J. P. Fishburn, “Clock skew optimization”, *IEEE Trans. Comp.*, vol. 39(7), July 1990, pp. 945-951.
- [9] A. Goldberg, “An efficient implementation of a scaling minimum-cost flow algorithm”, *J. Algorithms* 22, 1997, pp. 1-29.
- [10] A. Goldberg, *Network optimization library*. (Software tools) <http://www.avglab.com/andrew/soft.html>
- [11] M. Hutton and J. Pistorius, *Altera QUIP benchmarks*. <http://www.altera.com/education/univ/research/unv-quip.html>
- [12] A. Kuehlmann and J. Baumgartner, “Transformation-based verification using generalized retiming”, *Proc. CAV '01*.
- [13] C. E. Leiserson and J. B. Saxe. “Retiming synchronous circuitry”, *Algorithmica*, 1991, vol. 6, pp. 5-35.
- [14] N. Maheshwari and S. Sapatnekar, “Efficient retiming of large circuits”, *IEEE Trans VLSI*, 6(1), March 1998, pp. 74-83.
- [15] P. Pan, “Continuous retiming: Algorithms and applications”. *Proc. ICCD '97*, pp. 116-121.
- [16] S. S. Sapatnekar and R. B. Deokar, “Utilizing the retiming-skew equivalence in a practical algorithms for retiming large circuits”, *IEEE Trans. CAD*, vol. 15(10), Oct.1996, pp. 1237-1248.
- [17] N. Shenoy and R. Rudell, “Efficient implementation of retiming”, *Proc. ICCAD '94*, pp. 226-233.
- [18] D.R. Singh, V. Manohararajah, and S.D. Brown, “Incremental retiming for FPGA physical synthesis”, *Proc. DAC '05*, pp. 433-438.
- [19] H. J. Touati and R. K. Brayton, “Computing the initial states of retimed circuits”, *IEEE Trans. CAD*, vol. 12(1), Jan 1993, pp. 157-162.

# Formal Verification of Partial Good Self-Test Fencing Structures

Adrian E. Seigler, Gary A. Van Huben, and Hari Mony

**Abstract**— The concept of applying partial fencing to logic built-in self test (LBIST) hardware structures for the purpose of using partially good chips is well known in the chip design industry. Deceptively difficult though is the task of verifying that any particular implementation of partial fencing logic actually provides the desired behavior of blocking down-stream impact of all signals from fenced interfaces, and also ensuring that the partial fencing does not inadvertently preclude any common logic from being fully tested.

In this paper we discuss a case study for a verification method which exploits the power of formal verification to prove that any given partial fencing design satisfies all behavioral expectations. We describe the details of the verification method and discuss the benefits of using this approach versus using traditional simulation methods. We also discuss the testbenches created as part of applying this new method. Furthermore, we discuss the formal verification algorithms that were employed during application of the method along with the tuning that was done to enable efficient completion of the verification tasks at hand.

**Index Terms**— fencing, formal verification, self test

## I. INTRODUCTION

Logic Built-In Self Test (LBIST) [1],[2] is an inherent part of today's chip design and fabrication process. With the increasing density of chip die, it's now routine to implement multiple self contained units, cores or even systems within a single physical chip boundary. The most prominent example in the industry is multiple processor cores on a single CPU chip [3]. Such an example is shown in Figure 1. The so-called common logic in the figure comprises elements shared by the cores such as cache and logic for maintaining system coherency. With chips of this nature, running LBIST on the entire chip is adequate to ascertain whether the whole die is functional. However, to enable the use of a partially good chip[4] (such as a case where one or more cores are damaged and all the damage is contained within the boundaries of the damaged cores), the design must implement the concept of partial LBIST [5] fencing.

Partial LBIST fencing allows for self contained areas of a chip to be electrically isolated from the remainder of the chip in cases where such areas are damaged. In this manner, a procedure is employed such that if the LBIST of the entire chip indicates damage, then partial fences can be used to

electrically quarantine the affected regions. The partial LBIST signatures will be repeatable assuming that the remainder of the chip is functional and that the partial fencing is properly implemented.

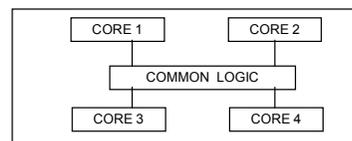


Fig. 1. Multiple core chip with common logic.

Since the risk exists that the interfaces connecting the damaged regions to the remainder of the chip are electrically unpredictable, it is imperative that the partial fencing be implemented correctly. Otherwise, a single missing fencing gate on an interface signal could result in unpredictable signatures. This would in turn result in perfectly usable partial good chips being discarded in the fabrication process.

There are several challenges in developing a robust and easy to use methodology for verifying partial LBIST fencing. The methodology has to scale to be applied on multiple-core chips with hundreds of millions of gates. This places restrictions on the tools that can be used and the algorithms that can be applied. Moreover, the methodology should not depend on having knowledge of the specific LBIST sequences used on the chip. Finally, the methodology should eliminate the need for verification engineers to spend significant time writing drivers and assertions, yet provide full coverage of the design to be tested.

The need for scalability, portability, and full coverage make traditional approaches like simulation [6] unattractive for the purpose of verifying partial LBIST fencing. It has already been shown that proper combination of formal and semi-formal algorithms can enable large-scale pervasive logic verification [7] using formal reasoning. Note however that prior work in this area does not cover verification of partial fencing. With these motivating factors in mind, a new and vastly superior methodology was developed and applied to an IBM z-Series multi-core chip. The description and results of this new approach are discussed here as we present a highly automated, scalable, reusable, and flexible methodology that performs functional formal verification using sequential equivalence checking to verify partial LBIST fencing.

Manuscript received May 8, 2007.

Adrian E. Seigler, Gary A. Van Huben, and Hari Mony - all from IBM Corporation.

## II. METHOD CONCEPTS

### A. Verification using Sequential Equivalence Checking

We selected the IBM semi-formal and formal verification tool SixthSense [8] as our primary tool for verification. It has already been shown that SixthSense is highly scalable with ability to verify designs with more than 100,000 state elements in the cone of influence. SixthSense provides various ways to abstract the design, including *blackboxing*, which is essential to enable application of our methodology on chip-level models. Moreover, SixthSense provides a wide variety of transformation algorithms for iterative simplification as well as verification algorithms for falsification and proofs.

To ensure portability and a high degree of automation, our methodology uses sequential equivalence checking [9] to perform verification of partial LBIST fencing. The verification objective is to ensure that all interface signals from the damaged portions of the chip are properly fenced. To set up the sequential equivalence checking run, we first develop two models of the design. Model 1 includes the original design with all the fence and interface signals driven to an inactive state. Model 2 includes the original design with all fence signals driven to an active state and all interface signals driven to non-deterministic values. In a typical sequential equivalence checking run, the outputs of the two designs are checked for equivalence. Since we needed to ensure that partial LBIST fencing worked properly, we configured the equivalence checking run to check the Multiple Input Signature Registers (MISR) [2] within the LBIST portion of the design for equivalence. In effect, the MISR is the output of LBIST as it contains the resulting signature from self-test.

There are several advantages in using sequential equivalence checking for partial LBIST verification. First of all, no dedicated manual effort is required to create the model or write assertions. Secondly, there is no need for complex *drivers* representing input assumptions. Third, it ensures portability; the only project specific items are identification of interface and fence signals and the MISR registers for checking equivalence. Finally, equivalence checking algorithms may be tuned for high scalability [11].

### B. False Failures due to Scan Chain Inversions

It is important to understand that if inversions in any of the latches comprising any of the MISR scan chains are allowed, then it is possible for the equivalence check described earlier to produce false failures. Figure 2 illustrates one example of how this could occur.

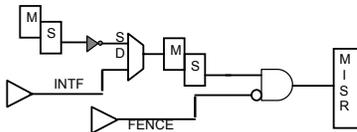


Fig. 2. Example of scan chain inversion using master-slave latches

The shaded inverter in the scan path will have the effect of causing a “0” from the first latch to be received as a “1” on the scan input of the second latch. Since the Model 1 mode of operation is such that all fence and interface signals are driven

to an inactive state (i.e. a logical “0” assuming positive active logic), this means that a “1” received by the second latch will subsequently be propagated into the MISR. However, since the same MISR bit is guaranteed to be “0” in Model 2 due to the fact that the fence bit is always active in Model 2, the two models would therefore not be equivalent. The mismatch is not the result of an improperly implemented fence, but is due solely to the scan-chain inversion. Note that this type of mismatch can occur only when the system is in scan mode, thus causing the multiplexer in Figure 2 to select scan signal S instead of data signal D.

In the design used in this case study, the design under test (DUT) contained no scan chain inversions. Nonetheless, the general method described here is equipped to handle scan chain inversions. This can be done for example by using a scan-chain traversal program to identify all inverting latches in the scan chain and then creating a scan input override list that would be used by SixthSense to ensure that the scan inputs to all the inverting latches are always driven to an inactive state. Note that by overriding scan inputs, the potential exists to mask bugs in the implementation of the partial LBIST fencing logic. To avoid such problems, we propose to make use of ternary (0,1,X)-logic [10]. More specifically, prior to executing any scan input overrides, we drive X values to all interface signals and make use of SixthSense in ternary mode to check whether it is possible to bleed X states into the Model 2 MISR. If such bleeding exists, the design bug(s) causing the bleeding must first be fixed before proceeding to override any scan inputs.

### C. Overfencing

The intended use of partial fences is to ensure that interface signals from partial good interfaces do not bleed into MISRs. This is illustrated in the upper portion of Figure 3 where the AND gate in the top of the figure serves to block the interface signal from being propagated downstream into sequential logic and the MISR. However, in an improperly implemented design, it is possible for overfencing to exist such that a fence signal also blocks common logic from impacting downstream logic. Figure 3 shows such an overfencing situation where the same fence signal that is used to properly block the partial good interface signal is also used to block common logic (via the AND gate in the bottom of the figure).

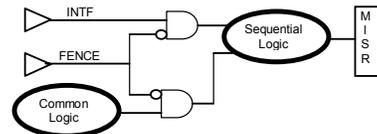


Fig. 3. Overfencing scenario.

The result of overfencing is that it prevents self-test of common logic, thus allowing for the possibility that any hardware faults that might exist in common logic will go undetected. So, although MISR signatures would be repeatable in circumstances where there is overfencing, the overall self-test behavior is not the desired behavior. Such a design flaw

would not easily be found using traditional validation methods like simulation, but would be quickly found using the equivalence check based verification method used in this case study. Note that if a design contains over fencing bugs, the ternary check for X-state propagation described earlier is not sufficient to detect this.

### III. METHOD DESCRIPTION

#### A. Method Steps

The method steps are as follows. Since there were no scan chain inversions in the design used for our case study, Steps 4, 6, and 7 were not required. Nonetheless, all steps for the general method are documented here for completeness.

1. Identify all partial good interfaces for the design whose partial fencing structure is to be verified. Referring to Fig. 1, the chosen interfaces would be the signals connecting the cores to the common logic. For every interface that connects a partial good component with common logic, identify the various control, address and data signals, along with their associated fence signals.
2. Create a wrapper whose inputs and outputs are the previously identified partial good interface signals. The wrapper schematic includes all common logic design components, and excludes all partial good components.
3. Create Model 1 and Model 2 drivers. The Model 1 driver forces all fence and interface signals to an inactive state. The Model 2 driver forces all fence signals to an active state and the interface signals are permitted to assume non-deterministic values.
4. Write an explicit assertion which checks for the propagation of X-states into the Model 2 MISR. Note that this assertion is only meaningful for Model 2 because the partial good interface signals in Model 1 are always inactive and thus are never driven with “X” values.
5. Execute the equivalence check to test that the Model 1 MISR is equivalent to the Model 2 MISR. In parallel, execute checking of the Model 2 X-state assertion. If no failures occur and there are no scan chain inversions, verification is complete. If failures occur with no scan chain inversions, the failures are design problems. If there are scan chain inversions, follow Steps 6 and 7 below to circumvent possible false failures. False failures are indicated by the presence of an equivalence check fail but no fail on the Model 2 X-state assertion.
6. Generate a list of inverted latches in the scan chain and force scan inputs to these latches to the inactive state.
7. Rebuild models and rerun the equivalence check. Any remaining failures are design problems. If no failures occur, verification is complete.

As mentioned already, Step 4 is not required if there are no scan chain inversions in the design. However, it is worthwhile to note here that in addition to the built-in equivalence checking of outputs or selected internal

signals, the SixthSense equivalence check mode also provides the ability for users to implement and test customized assertions on either model. Exploiting this capability provides the method with a means to handle potentially false failures that can be produced from the equivalence check when there are scan chain inversions since any failures on the Model 2 X-state assertion are accepted to be design problems and must be rectified.

#### B. Advantages of the Method

There are several benefits to using the above-described method as compared to traditional simulation based approaches. These benefits include:

- **Proof of Correctness** - Our method proves no missing fences and no over fencing. Since it is not practical to exhaustively exercise all combinations of inputs and internal states via simulation in most real world designs, this precludes the possibility of obtaining proofs for the verification properties of interest using simulation.
- **Scalability** - Verification complexity using our method does not increase significantly with complexity or size of the DUT. Unlike simulation, the cost to setup and execute this method is not influenced greatly by the size of the design. This case study shows that application of the method to composite models with more than a million latches is indeed realizable and practical. Attempting to verify the design in this case study using simulation would have consumed considerably more time and would have been inherently incomplete.
- **No knowledge of LBIST sequences required** – Despite the fact that the LBIST sequences used in actual hardware are generally complex sequences that require scanning, our method obviates the need to develop complex drivers as well as the need to manage and update drivers if and when LBIST sequences change. In contrast, simulation often poses the need for one to develop relatively complex testbenches in order to avoid driving invalid input vector combinations and to ensure “interesting” and “corner case” scenarios are tested. By using this formal verification based method, all possible combinations of inputs allowed by the testbench are automatically covered.
- **Setup is very easy** - Once the fence signals and partial good interfaces have been identified (per Step 1 of the method), the verification setup can be auto-generated using scripts.
- **No need for complex assertions:** – Equivalence checking mode is already built into the SixthSense tool, so there is no need to write explicit assertions to verify correctness. The single explicit assertion to check for X-state propagation in Model 2 is trivial to implement.
- **Method supports any partial good self test structure** – The process for creating the composite testbench is the same regardless of design implementation details. Even though it is possible for designs that contain scan

chain inversions to initially produce false failures, such failures are easily identified and subsequently filtered out of the final verification results.

#### IV. VERIFICATION RESULTS

The proposed verification methodology was applied to an IBM z-Series multi-core chip. Due to the hierarchical nature of the design that was verified, the verification was carried out in two parts – one part at the unit level, and the other part at the chip level. At the unit level, there were two processor cores sharing common logic. At the chip level, there were two units instantiated on the chip with another set of common logic being shared by these two instances. To verify the partial LBIST fencing, one only needs to analyze the common logic shared at the unit and chip levels. Before we built our models, we abstracted all the irrelevant logic using blackboxing. Our application of the methodology was highly successful as we were able to identify a total of 6 design bugs. After these bugs were fixed, we were able to prove that partial LBIST fencing worked properly. Table 1 summarizes the major verification metrics for each of the two models tested.

TABLE 1  
VERIFICATION SUMMARY

Verification Metric	Core Level Model	Chip Level Model
# Inputs (thousands)	6.1	41
# Gates (millions)	2	24
# Registers (millions)	2.1	2.8
Run Time	639 sec	1654 sec
Peak Memory Usage	6.8 GB	16.7 GB
# Design Bugs Found	2	4

#### V. TUNING THE VERIFICATION RUNS

Our two primary challenges when tuning the SixthSense sequential equivalence checking algorithms for verification of partial LBIST fencing were to **1)** find bugs in the design as fast as possible and **2)** efficiently complete proofs of correctness. To address both challenges we used the **EQV** engine, a sequential redundancy removal engine that uses an *assume-then-prove* paradigm to identify and merge gates that are sequentially redundant [11]. The first step is to guess redundant gates using a variety of techniques such as semi-formal analysis, structural analysis, name comparisons, etc. A speculative merge of the redundancy candidates is then performed to create the model to be equivalence checked (the “assume” step). Finally, proof analysis is performed on the speculatively-reduced model to attempt to validate the correctness of redundancy candidates (the “proof” step).

It is well known that SAT-based *bounded model checking* (BMC) is one of the best approaches for falsification. However, we quickly realized that for the sizes of the designs we were interested in verifying, SAT-based BMC quickly runs out of steam. We decided to follow the approach of [11] and applied SAT-based BMC on the speculatively-reduced model for falsification. Speculative-merging enabled deeper BMC and we found all the bugs in the design using this approach. However, LBIST sequences are very long and typically take hundreds of cycles to update the MISRs making it practically

impossible to develop a good level of confidence about the design given a few hundred cycles of BMC. This makes it imperative that proofs of correctness be obtained.

The wide variety of synergistic transformation and verification algorithms available in SixthSense were very useful in validating the correctness of redundancy candidates ; Localization in particular was found to be very effective. The SixthSense feature that proved most useful is the ability to identify *causal* redundancy assumptions that make proofs difficult. This is a powerful feature as it enabled the verification engineers to identify certain constraints that the design must satisfy to prove the validity of the causal redundancy assumptions. By adding assertions to check for the satisfaction of the constraints, we were able to figure out illegal driving of certain signals in the design and fine-tune the driver accordingly. Note that the only other way we could have found the illegal driving would be through falsification showing that the MISRs differ. Due to the large fail depth, even BMC on the speculatively-merged model was not successful.

#### VI. CONCLUSION

The application of our partial fencing verification approach to an IBM z-Series multi-core chip provides solid evidence of ease of use, scalability, and flexibility of the methodology. Six design problems were found and resolved prior to initial chip release. These problems would have almost certainly not been found via simulation, and would have hampered the ability to reliably identify and use partial good chips in real systems. We therefore advocate not only the continued use of this methodology going forward, but also strongly encourage the continued exploration and application of formal verification to other non-trivial verification tasks .

#### REFERENCES

- [1] G. A. Van Huben, “The role of two-cycle simulation in the s/390 verification process,” *IBM Journal of Research and Development*, vol. 41, no 4/5, 1997.
- [2] W. V. Huott et al., “Advanced microprocessor test strategy and methodology,” *IBM Journal of Research and Development*, vol. 41, no 4/5, 1997.
- [3] J. A. Kahle et al., “Introduction to the Cell microprocessor,” *IBM Journal of Research and Development*, vol. 49, no 4/5, 2005.
- [4] L. Farnsworth et al., “Partial good integrated circuit and method of testing same”, in *U.S. Patent US20050047224A1*, 2005.
- [5] M. Riley et al., “Testability Features of the First-Generation Cell Processor,” in ITC, Nov, 2005
- [6] B. Wile, J. Goss, W. Roesner, *Comprehensive Functional Verification*, San Francisco, CA, Elsevier, pp. 141-197, 439-485.
- [7] T. Glokler et al. “Enabling Large-Scale Pervasive Logic Verification through Multi-Algorithmic Formal Reasoning,” in FMCAD, Nov. 2006
- [8] H. Mony et al., “Scalable Automated Verification via Expert System Guided Transformations”, FMCAD 2004
- [9] A. Kuehlmann, C. van Eijk, “Combinational and Sequential Equivalence Checking”, in *Logic Synthesis and Verification*. Kluwer Academic Publishers, 2004.
- [10] A. Jain et al., “Testing, verification and diagnosis in the presence of unknowns”, in VLSI Test Symposium, 2000.
- [11] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, “Exploiting Suspected Redundancy without Proving It”, DAC 2005.

## Case study: Integrating FV and DV in the Verification of the Intel<sup>®</sup> Core<sup>™</sup>2 Duo Microprocessor

Alon Flaisher, Alon Gluska, Eli Singerman  
*Intel Corporation, Haifa, Israel*  
(alon.flaisher, alon.gluska, eli.singerman)@intel.com

### Abstract

*The ever-growing complexity of Intel<sup>®</sup> CPUs, together with shortened time-to-market requirements, poses significant challenges for pre-silicon logic verification. To address the increasing verification gap, major improvements to verification practices are required. In Merom, the Intel<sup>®</sup> Core<sup>™</sup>2 Duo microprocessor, we integrated Formal Verification (FV) with Dynamic Verification (DV) such that FV was also practiced by non-FV experts and replaced some traditional, simulation-based verification activities. This led to both higher productivity and better quality compared to previous projects. In this paper we report on the integration we used, including two examples, results, and future directions.*

### 1. Introduction

Formal verification (FV) is widely known as an effective verification method, and has been successfully applied to a wide variety of designs, including software [3], and in particular in the verification of several microprocessors [4]. In most cases, FV is performed by FV experts separately from traditional simulation-based dynamic verification (DV), providing full proofs for areas of high risk [1][7][8]. FV has proven able to find extreme corner-case bugs, and it provides a high degree of confidence that such bugs do not exist. Nevertheless, typically FV has been applied only to a small fraction of the design. There are several reasons for this. First, FV is costly when done in addition to other simulation-based activities. In order to apply FV on a wide scale, a project needs a large FV team. This leads to the next problem, that of finding experienced FV engineers who can carry out complex proofs in a reasonable amount of time. Next is the relatively limited capacity of FV tools, which typically cannot handle large designs such as the functional units of an industrial microprocessor.

In this paper we present our experience in the effective utilization of FV techniques in the verification of Merom, the Intel<sup>®</sup> Core<sup>™</sup>2 Duo microprocessor. We integrated FV and DV by providing DV engineers with basic skills and user-friendly tools to perform FV tasks and by replacing simulation-based activities such as coverage or stress with FV. DV engineers, in addition to FV engineers, owned the identification, specification and implementation of FV tasks

in their areas of responsibility. This approach overcame some of the problems mentioned above. First, typically the verification engineer is familiar with the design. Second, the verification engineer executes both FV and DV and can identify what is covered by each activity. Finally, there is more flexibility in assigning engineers to FV tasks. This approach tightly linked FV with DV and reduced overall verification costs while achieving high-quality results. One example presented in this paper, in which the results exceeded our expectations, is the execution cluster (EXE), where all arithmetic operations are implemented. By checking every possible combination of inputs on thousands of micro-operations (uops), we achieved full data-path verification of nearly the entire cluster. These results are the direct outcome of the FV/DV approach, as EXE DV engineers used FV and completely dropped functional coverage and other simulation-based techniques.

This paper is organized as follows: we start with a review of related work, then we describe the FV/DV approach and provide examples of its application to a real-life microprocessor, and finally we describe lessons learned and future challenges.

### 2. The FV/DV Approach

The elements of the FV/DV approach are: deciding where to apply FV (based on the limitations of simulation); having joint test plans; replacing simulation activities with FV; having verification engineers from the simulation teams also applying FV; establishing (by FV experts) FV environments for verification engineers to use; having joint checking; tool integration. In this section, we elaborate on several key elements.

**Deciding Where to Apply FV.** The verification teams identify areas where simulation is less efficient - typically designs with a large data space or areas with low controllability in simulation. FV experts help examine whether FV can be applied efficiently to these areas by running feasibility checks. These feasibility checks include identifying the main properties to be verified, examining the size and complexity of the interface, adding basic

assumptions such as the behavior of clock and reset, and running several witnesses in order to estimate the maximum bound that the BMC (Bounded Model Checking) tools can reach on this design.

**Effect on Simulation.** The next element, which enables shifting DV resources to FV activities and improves the FV return on investment, is the effect of FV on DV. For every design under test, one decides if FV should completely replace DV, replace DV stress or coverage activities, or complement DV. Because DV coverage activities usually require a considerable effort (writing and reaching coverage goals), replacing them with FV increases the FV return on investment. However, it is not always possible to fully eliminate DV activities in favor of FV, as FV often covers only *part* of the design. When FV is applied to a block, it can replace DV activities that are targeted in checking that block. For example, if FV is verifying an Adder unit, there is no need to cover different data scenarios in DV. On the other hand, if DV coverage effort is applied to a specific unit only for making sure that the overall system exhibits interesting behaviors, FV should not replace it. While the decision to drop simulation activities is easier when a full FV proof is in place, it can also be made in the case of bounded proofs when the confidence level obtained from these is sufficient.

**Running FV by Verification Engineers.** The next element of the FV/DV approach is having FV performed not only by FV experts, but also by verification engineers. In some cases, FV activities can be performed entirely by the verification engineer, technically supported by the FV team. In other cases, the verification activity is divided between the FV expert and the verification engineer. In the latter case, the FV expert establishes a formal environment and the verification engineer develops the specifications (FV checkers) within this environment. This method of dividing FV between the FV expert and the verification engineer is especially useful when FV skills are required to establish a working environment in which specifications can then be written in a straightforward manner. In the previous section we listed several advantages to performing FV activity in the DV verification teams, but of course, there are also disadvantages. FV tools are sometimes more difficult to use and require a different mindset than simulation. Some proofs require decomposition, induction, or abstraction, which require FV expertise. Therefore, when assigning an engineer to an FV task, one should consider how complex the design is vs. how complex the proof is expected to be.

**Joint Checking.** The next element is joint checking for simulation and FV. This is mainly achieved through the use of RTL assertions [20] which run both in FV and in simulation flows. All assertions and assumptions (environment restrictions) used by the FV proofs are also

used as checkers in simulation. In addition, FV checkers re-use assertions or synthesizable reference models that have been developed for simulation.

### 3. Application of the FV/DV Approach to the Intel® Core™ 2 Duo Microprocessor

The strategic decision to integrate FV and simulation in the verification of the Intel® Core™ 2 Duo microprocessor was taken in the early planning stages. Rather than have FV be orthogonal to simulation activities, we decided to incorporate FV as yet another logic verification capability, following the approach described in the previous section. We now present the results of applying this to two examples: the execution cluster (hereafter referred to as EXE) and the Micro-Instruction Sequencer unit (hereafter referred to as MS). EXE consists of all the arithmetic and logic units, and FV was performed using symbolic trajectory evaluation (which is a form of symbolic simulation) [9], under Intel's FORTE system [10], in a joint effort between the formal and EXE verification teams. The FV of the MS unit was carried out using BMC [5] with specs in the ForSpec language [2] by a verification engineer from the simulation-based team. The results in both examples exceeded our expectations, and were a direct consequence of the FV-DV approach.

#### A. FV of the Entire EXE Cluster

The EXE (ALU of the Microprocessor) cluster work was divided such that an FV expert developed the Cluster Formal Environment (hereafter referred to as CFE) platform and the EXE verification engineers formally verified their specifications in this platform. Although symbolic simulation is different than standard simulation, it is similar conceptually and is therefore rather natural for verification engineers. The major difference between a standard RTL simulator and a symbolic one is that the latter runs all possible traces together and does not need concrete tests as stimuli. A symbolic simulator computes the value of circuit nodes (e.g., outputs) at given time-ticks as a function of the inputs. To use symbolic simulation, one has to provide symbolic stimuli. After the simulator computes the function of the circuit, one has to compare the results against the desired output (also expressed as a function of the inputs). If the functions match, we conclude that the circuit meets the specification. In case of a mismatch, one has to debug and analyze whether the problem is in the specification or in the circuit.

The EXE cluster consists of several relatively independent units which perform the different arithmetic and logic operations. This made it possible for the FV expert to develop a symbolic simulation framework driving the units one by one, placing symbolic input variables on the cluster interface and coping with capacity problems.

When an operation in a specific unit has been simulated, inputs of the remaining units were driven by X's, indicating don't care values. Propagating X's is much cheaper for the symbolic simulator, and hence it was able to simulate the entire EXE cluster. It would have been impossible to simulate the entire cluster symbolically without driving one unit at a time and abstracting all other units with don't care values.

The verification unit owners helped the FV expert build the portion of their unit in the CFE and performed the verification by writing and running FV checkers for micro-instructions (uops) in their units. These checkers were developed by the EXE verification engineers in FL (Functional interpreted programming Language). This language enables one to write functions operating on symbolic arguments. Learning FL is not trivial, but nevertheless the engineers were able to code simple specifications within a couple of weeks, and managed to capture all required specifications.

EXE FV activity was carried out in the second half of the project, whereas standard simulation was used earlier for basic testing. In this stage (the second half of the project) we performed full verification, reached all corner cases, and acquired the confidence level required for tape-out. The EXE engineers used FV as the main verification tool for this stage *instead* of the massive *coverage* work which was used in previous projects. They gradually expanded their FV activities until it had been performed on all units, formally verifying the vast majority of all EXE micro-operations. We should note that several highly complex proofs, such as the divider, were carried out separately by FV experts.

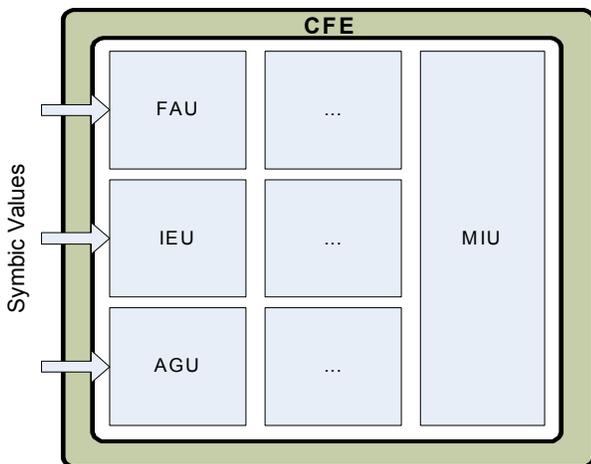


FIGURE 1: ARITHMETIC UNITS VERIFIED BY CLUSTER ENGINEERS IN THE CLUSTER FV ENVIRONMENT DEVELOPED BY THE FV TEAM

Overall, results exceeded our expectations. By checking every possible combination of inputs on thousands of uops, we achieved full datapath verification of nearly the entire

cluster. Overall, 98% of the uops were verified, far above any CPU verification project at Intel. These results are the direct outcome of the strategic FV/DV approach whereby EXE DVers wrote FV specifications and ran FV using symbolic simulation. We strongly believe that these DV engineers, who had no previous background in FV, achieved a much higher quality of verification using FV than they had in previous projects using DV with a similar effort.

### B. FV as the Main Verification Tool for the MS Unit

The FV of the MS unit is yet another example of the advantages of FV over simulation-based verification. Here FV completely replaced planned DV tasks (test plan writing, test writing, coverage, etc). The MS unit holds a ROM of micro-instructions (a.k.a. uops) which it sequences as part of the translation of IA instructions into sequences of uops. This is done using fairly complex code that needs to support all the combinations of uops that may co-exist in the ROM under all possible events. The large testing space exists only partially in any given realization of the ROM, while verification needs to completely verify the hardware for any possible realization. It was well understood that traditional DV would fall short of high-quality verification. Yet we identified that the unit meets the FV BMC criteria.

The first FV model was developed by the cluster verification team within about 3 months, and targeted the dependencies between neighboring uops and between uops and events. A reference model for the MS unit was developed in ForSpec. It modeled the entire functionality of the unit under normal operation, excluding some specifics such as counters that could not fit into the FV tools. A higher abstraction level reduced the chance of errors and made the reference model robust and insensitive to non-functional micro-architecture changes. All the assumptions of the FV model were checked in simulation. Another month was required to complete 18 properties covering the complete external behavior of the MS unit under all possible combinations of uops and external events. The model held approximately 1400 state variables, and BMC runs reached a bound of about 40 clock phases. It was analyzed and confirmed that all possible sequences within the MS unit can be reached within 28 phases, including the reset sequence. Indeed, all the bugs found by the FV model were below this bound. Therefore, a bound of 40 phases was confirmed as sufficient for any applicable scenario in the MS unit.

Almost all the logic in the MS unit was verified solely using FV, and the functionality that was formally verified was *completely* dropped from the simulation-based space. No planned DV effort was carried out in the MS unit except for the testing of one certain type of instruction for which new logic had been added and which did not fit into the FV

model. Not a single test was written, and no coverage monitor was coded. In summary, FV detected 18 RTL bugs, including several high quality bugs which most likely would have escaped any simulation activity and would reach silicon. In the MS unit, FV was used as the main verification tool, providing superior quality and saving the considerable amount of effort it would have taken to develop a comparable DV environment. Once FV was complete it was used by the designers as a means of preventing any further bugs from being introduced into the RTL model.

### C. Additional Results

FV was successfully used as part of the verification of a dozen modules in the Intel® Core™2 Duo microprocessor, consuming overall almost 10% of the effort invested in verification. Overall, 75% of the proofs developed impacted simulation activities, mainly by replacing functional coverage activities. Almost half the proofs were done by FV engineers, and the rest by verification engineers in the DV verification teams.

## 4. Conclusions and Recommendations

We applied a new approach for using FV in the verification of the Intel® Core™2 Duo microprocessor, in which FV became an integral part of the verification capabilities. We incorporated FV as yet another logic verification capability and enabled many of the FV activities to be performed by engineers from the verification teams. They used bounded model checking for most proofs, but also made use of symbolic model checking and symbolic simulation. In most areas, results exceeded our expectations. In the EXE cluster, in which all arithmetic and logic operations are implemented, we achieved full datapath verification of nearly the entire cluster by checking every possible combination of inputs on thousands of micro-operations (uops). EXE verification engineers used FV in a formal environment developed by an FV expert, and completely dropped functional coverage.

In the MS unit, FV took over simulation completely. Almost the whole unit was modeled in ForSpec and BMC fully covered all possible scenarios. Once it became available, the FV model found a whole range of bugs, some of which could not be practically found by simulation. In both examples, FV activities were performed by engineers from the verification teams who were not FV experts, and FV provided a significantly higher level of confidence for a significantly lower effort compared to the simulation-based activities they replaced.

The Intel® Core™2 Duo microprocessor was the first microprocessor project in Intel to deploy such integration between simulation and FV. We believe that this approach provides a means of utilizing verification resources more

effectively. We will strengthen the FV/DV approach in our next generation microprocessor project. We have taken additional steps to further integrate these two activities, mainly via major simplifications of the tool user interface, a new assertion- based verification (ABV) methodology that facilitates the deployment of FV proofs, and broader expertise in the use of FV within the verification team. We plan to make more extensive use of FV for bug-hunting in the early stages of the project.

## 5. Acknowledgements

We thank Nissim Aharon, Gavriel Gavriellov, Uri Juhasz, Dan Carmi, Micky Hadash and Amit Gradstein for supporting and taking an active part in applying the FV/DV methodology. We also thank Miriam Brusilovsky, Paul Inbar, Orna Grumberg and Moshe Vardi for their valuable comments on an earlier draft of this paper.

## 6. References

- [1] M. Aagaard, R. B. Jones, R. Kaivola, K. Kohatsu, C.-J. Seger, Formal verification of iterative algorithms in microprocessors, DAC 37, 2000.
- [2] R. Armoni et. Al., The ForSpec Temporal Logic: A New Temporal Property-Specification Language, TACAS 2002.
- [3] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. *SPIN 2001*.
- [4] Bob Bentley. Validating the Intel Pentium 4 Microprocessor, In Proceedings of the 40<sup>th</sup> Design Automation Conference, 2003.
- [5] F. Coptly, L. Fix, R. Fraer, E. Guinchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. *CAV 2001*.
- [6] K. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer Academic, 1993.
- [7] D. M. Russinoff: A Mechanically Checked Proof of Correctness of the AMD K5 Floating Point Square Root Microcode. *Formal Methods in System Design* 14(1): 75-125 (1999).
- [8] T. Schubert, High level formal verification of next-generation microprocessors, DAC 40, 2003.
- [9] C.-J. Seger and R. Bryant, Formal verification by symbolic evaluation of partially-ordered trajectories, *Formal Methods in System Design*, 6(2):147--190, 1995.
- [10] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, An Industrially Effective Environment for Formal Hardware Verification, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9 (September 2005), pp. 1381-1405.
- [11] H. Foster, A. Krolnik, D. Lacey, *Assertion-Based Design*, Kluwer Academic Publishers, 2<sup>nd</sup> edition, 2003.

# Circuit Level Verification of a High-Speed Toggle

Chao Yan and Mark R. Greenstreet  
 Department of Computer Science  
 University of British Columbia

**Abstract**—As VLSI fabrication technology progresses to 65nm feature sizes and smaller, transistors no longer operate as ideal switches. This motivates verifying digital circuits using continuous models. This paper presents the verification of the high-speed, toggle flip-flop proposed by Yuan and Svensson [1]. Our approach builds on the projection based methods originally proposed by Greenstreet and Mitchell [2], [3]. While they were only able to demonstrate their approach with two- and three-dimensional systems, we apply projection based analysis to a seven-dimensional model for the flip-flop. We believe that this is the largest verification to date of a digital circuit using non-linear circuit-level models.

In this paper, we describe how we overcame problems of numerical errors and instability associated with the original projection based methods. In particular, we present a novel linear-program solver and new methods for constructing accurate linear approximations of non-linear dynamics. We use the toggle flip-flop as an example and consider how these methods could be extended to verify a standard cell library for digital design.

## I. INTRODUCTION

Deep-submicron technologies simultaneously confront designers with transistor behaviors that require circuit-level models to produce working designs and with integration densities that require working at high-levels of abstraction. Due to leakage currents, small transistors do not operate as the ideal switches that have been the foundation of synthesis and switch-level simulation tools for the past twenty years. Other deep-submicron challenges include crosstalk, power-supply noise, and random parameter variations. Current design tools do not provide a satisfactory way of dealing with these issues. Typically, coarse approximations or heuristics are used to validate designs, an approach that simultaneously sacrifices performance and fails to guarantee working silicon. Formal methods can help address these challenges by verifying proper behaviors at low-levels of abstraction and ensuring that the abstractions of these low-level behaviors to higher levels of abstraction are sound.

A circuit with a continuous model has an uncountably infinite state space, typically  $\mathbb{R}^d$  where  $d$  is the number of nodes in the circuit. Furthermore, the non-linear differential equations that describe circuit behavior do not have closed form solutions; thus, purely symbolic methods are unlikely to succeed for circuit-level verification. To address these problems, tractable approximation methods are needed. To ensure soundness, these approximations must over approximate the reachable space. While such approximation-based techniques cannot be complete, we want the approximations to be accurate enough that the verification method can be applied to real designs. Finally, the reachability computations must be

sufficiently efficient as to enable the verification of real designs in a tolerable amount of time.

The current work builds upon the projection based methods for a tool called “Coho” originally proposed in [2], [3]. Coho represents a region in  $\mathbb{R}^d$  by its projection onto two-dimensional subspaces. The intuition behind this approach is that the input/output behaviors of digital circuits can be captured by these projections that correspond to pairs of causally related variables.

The framework presented in [2], [3] was only applied to simple two- and three-dimensional models due to numerical conditioning problems that often occurred in solving the linear-programs that were used when calculating the reachable space. We present our solution to these numerical problems. We also describe places where the original formulation for Coho resulted in large over approximations and present refinements that enable much more accurate calculations.

We apply our modified version of Coho to the verification of a toggle flip-flop that was originally presented by Yuan and Svensson [1]. This toggle is a relatively simple circuit consisting of nine transistors and seven nodes. This makes it similar in size to many cells in a typical library for standard cell design. For example, if a design has a critical timing path that cannot be satisfied by gate sizing or local logical changes, a designer might identify a custom logic function that could be implemented to achieve an acceptable delay. In current design flows, adding a cell to the library is an arduous task, in large part due to the large number of circuit simulations that must be manually run and checked; the time required to perform such simulations is often unacceptable for the project schedule. Thus, a design team in this situation can be forced to either make large architectural changes or to lower performance targets. Having verified the toggle, we can realistically hope to verify cells that a designer wants to add to a cell library. Similar opportunities to improve the design flow arise when trying to satisfy cross-talk requirements, when using dynamic or highly-skewed logic gates, when integrating self-timed circuits in a synchronous design, etc.

The toggle is an attractive first example for circuit-level verification because it has an interesting sequential behavior that we can specify very precisely in both discrete and continuous formulations. It exhibits a critical race which places demands on the accuracy of the approximations made by any reachability tool. Finally, with seven nodes, it presents an opportunity to verify discrete properties of a seven-dimensional, non-linear system. We are not aware of any previous examples of verifying digital circuit behaviour from non-linear circuit

models for circuits of more than four nodes.

## II. RELATED WORK

The problem of verifying systems with a combination of discrete and continuous models has been a focus of the hybrid systems research in the past ten years. This section first examines prior work in verifying circuits and hybrid systems and then examines the prior work on Coho upon which our current research is built.

### A. Verification of Hybrid Systems

One of the earliest tools for verifying hybrid systems was *HyTech* [4], [5]. Based on *linear hybrid automata* [6], *HyTech* models continuous variables as piecewise linear functions of time. While non-linear systems can be approximated by piecewise linear ones [7], the number of pieces required to obtain a given degree of accuracy grows exponentially with the dimensionality of the system, exacerbating the high complexity of the model-checking algorithm. Even with the simplistic assumptions that each variable evolves as a piecewise linear function of time, nearly all properties of linear hybrid automata are undecidable [8], [9]. Thus, heuristics and approximation methods are required for verifying real circuits.

The ideas in *HyTech* have been extended recently by Frehse in the implementation *PHAVer* [10]. *PHAVer* uses arbitrary precision integer and rational arithmetic to avoid problems of numerical overflow that limited *HyTech*. Furthermore, *PHAVer* uses the Parma Polyhedra Library [11]. *PHAVer* has verified larger problems than *HyTech* could including a non-linear, tunnel-diode oscillator circuit and a VCO [12]. We are not aware of any applications of *PHAVer* to digital circuit designs.

The *d/dt* tool [13] performs reachability analysis of continuous or hybrid systems modeled by linear differential inclusions of the form  $dx/dt = Ax + Bu$ , where  $u$  is an external input taking values in a bounded convex polyhedron. *d/dt* represents the reachable sets as non-convex orthogonal polyhedra [14], i.e. finite unions of full-dimensional, fixed size hyper-rectangles, and approximates the reachable state using numerical integration and polyhedral approximation. To the best of our knowledge, all examples computed with *d/dt* have been low, dimensional, with two- or three-dimensions, which we believe reflects the high-complexity of representing the reachable space as an explicit set of hypercubes. Earlier, Kurshan and McMillan [15] used a similar approach, for the verification of a simple arbiter, a four-dimensional, non-linear problem.

*CheckMate* [16] is a Matlab based tool for modeling, simulating and verifying properties of a class of hybrid systems: *threshold-event-driven hybrid systems*. *Checkmate* can model systems with non-linear dynamics by computing a convex polyhedral approximation of the reachable region. *Checkmate* has recently been used to verify simple circuits including a tunnel-diode oscillator and a Sigma-Delta modulator [17], both modeled with three-dimensional state spaces. As with *PHAVer*, we are not aware of any applications of *Checkmate* to digital circuit designs or to circuits with higher dimensional models.

### B. Coho

Coho represents reachable sets with *projectagons*. A projectagon is the high dimensional (and potentially nonconvex) bounded polytope formed by the intersection of a collection of prisms. Each prism is unbounded in all but two dimensions, and in those two dimensions the cross-section of the prism is a bounded polygon. The projection polygons are not required to be convex; thus, non-convex, high-dimensional objects can be represented by projectagons. The high-dimensional object represented by a projectagon is the set of all points that satisfy the constraints of each projection. As an example, Figure 1 shows how a three-dimensional object (the ‘‘anvil’’) can be represented by its projection onto the  $xy$ ,  $yz$ , and  $xz$  planes.

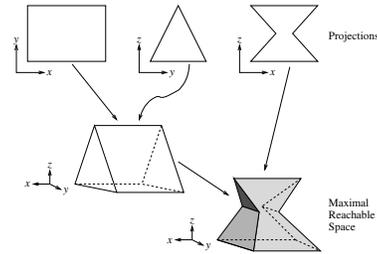


Fig. 1. A Three-Dimensional ‘‘Projectagon’’

The basic ideas behind Coho’s reachability analysis are straightforward. We require that the time derivatives for the state variables are finite and therefore that trajectories are continuous. This, the extremal trajectories emanate from the faces of the projectagon. At each time step in its reachability analysis, Coho derives a convex polytope that contains all points within a preset distance of the current projectagon. Coho then derives bounds on the time derivatives of the state variables within this polytope, and uses these to determine a time step size for which all trajectories are guaranteed to remain in the polytope. Coho then moves each face according to its maximum outward derivative to obtain a projectagon that contains the reachable space at the end of the time step. We describe these steps in greater detail below.

Let  $P_0$  denote the projectagon for the reachable space at the beginning of a time step, and let  $\lambda$  be a positive real. To derive a convex polytope that contains all points within distance  $\lambda$  of  $P_0$ , we observe that the convex-hull of a projectagon is contained in the projectagon obtained from the convex hulls of each of the projection polygons. Thus, Coho computes the hulls of these two dimensional polygons. Coho then moves all edges of these hulls outward by  $\lambda$ . These bloated hulls are the projection polygons of a projectagon that contains all points that are within distance  $\lambda$  of the original projectagon. We call this bloated version of the original projectagon  $P_{bloat}$ . Coho represents  $P_{bloat}$  as a linear program.

Faces of the projectagon correspond to edges of the projection polygons. Thus, the version of Coho described in [3] operated on one projection polygon edge at a time. We describe this approach here and describe our refinement of

this method in Section IV-C. For each polygon edge, Coho constructs an oriented rectangle that contains all points within distance  $\lambda$  of the edge. The intersection of the prism for this rectangle with  $P_{bloat}$  contains all points within distance  $\lambda$  of the current face. We call this region  $P_{bloat}$ , and Coho represents it with a linear program.

Coho then calls the user supplied circuit model with  $P_{bloat}$ . The model returns a matrix  $A$ , vectors  $b$ , and positive vector  $u$  such that

$$Av + b - u \leq \dot{v} \leq Av + b + u \quad (1)$$

for any point  $v \in P_{bloat}$ . Based on this linearized model and the linear program  $P_{bloat}$ , Coho constructs a linear program for all points reachable from the face at the end of the time step. Coho then projects the feasible region of this linear program back onto the basis variables for the polygon edge associated with the face to obtain the time advanced edge. Details are given in [3].

We now focus on those aspects of the Coho algorithm that we had to modify in order to successfully verify the toggle. The modifications that we made should be applicable to other circuit and hybrid systems verification problems as well. First, Coho makes extensive use of linear programs (LPs). Earlier experience with Coho showed that these LPs were often highly ill-conditioned, causing off-the-shelf linear program solvers such as Matlab's `linprog` and a direct implementation of Simplex to fail. In Section IV-A, we describe our robust linear program solver for the LPs that arise in Coho. Second, Coho was originally implemented with uniform bloats for all variables. We found that this led to both large error bounds when linearizing the circuit model and to Coho taking very small time steps. Section IV-B describes how we modified Coho to use different bloat amounts for different variables.

Each time a projection polygon edge is advanced, a new polygon is produced. The union of these "edge polygons" forms the boundary of the projection polygon at the end of the time step. This process can lead to a rapid increase in the number of projection polygon edges. Section IV-C describes how the original Coho simplified the projection polygons at each step to control this blow-up. In the same section, we present our modifications to the original algorithm to reduce the amount of over approximation at critical phases of the verification.

### III. THE YUAN-SVENSSON TOGGLE

Figure 2 shows the toggle circuit from [1]. Transistors are labeled with their shape factors and the capacitor on the  $q$  output represents a load equivalent to the gate capacitance of transistors with the a total shape factor of 36; this is the load that the toggle places on its clock input. We use this load to verify that the output of one toggle can drive the clock input of another to implement a ripple counter.

The operation of this circuit can be understood by using a simple switch model starting from a state where the  $\phi$  input is low. In this case,  $y$  is driven high,  $z$  is floating, and  $x$  is the logical negation of  $z$ . Figure 3 shows the state transition

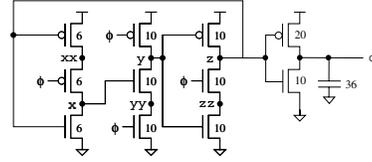


Fig. 2. Yuan and Svensson's Toggle Circuit

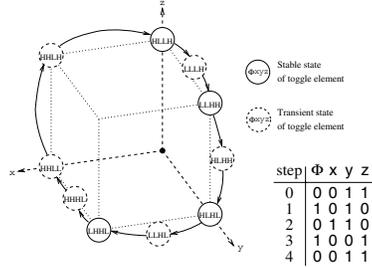


Fig. 3. State Transition Diagram for the Toggle

diagram for the toggle starting from the state where  $z$  is high when  $\phi$  is low – the other case, with  $z$  high, is reached on step 2 of the figure. Note that from step 2 to 3 in the figure, all three of  $x$ ,  $y$  and  $z$  change values. This is a critical race for the toggle. We further note that if the rise or fall times for  $\phi$  are too large, the toggle will fail.

To specify the desired continuous behavior of the toggle circuit, we use the Brockett annulus construction [18] shown in Figure 4. When a variable is in region 1, its value is constrained but its derivative may be either positive or negative. Thus, region 1 of the annulus specifies a logically low signal: it may vary in a small interval around the nominal value for low signals. When the variable leaves region 1, it must be increasing; therefore, it enters region 2. Because the derivative of the variable is positive in region 2, it makes a monotonic transition leading to region 3. Regions 3 and 4 are analogous to regions 1 and 2 corresponding to logically high and monotonically falling signals respectively. Because transitions through regions 2 and 4 are monotonic, traversals of these regions are distinct events. This provides a topological basis for discrete behaviors. Furthermore, the horizontal radii of the annulus define the maximum and minimum high and low levels of the signal (i.e.  $V_{0l}$ ,  $V_{0h}$ ,  $V_{1l}$ , and  $V_{1h}$  in

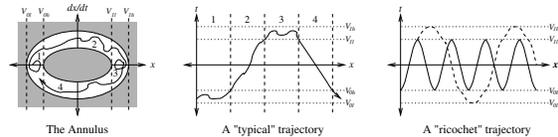


Fig. 4. Brockett's Annulus

figure 4). The maximum and minimum rise time for the signal correspond to trajectories along the upper-inner and upper-outer boundaries of the annulus respectively. Likewise, the lower-inner and lower-outer boundaries of the annulus specify the maximum and minimum fall times.

Note that a signal may remain in regions 1 or 3 arbitrarily long. This is essential when verifying the toggle where we must show that the output satisfies the constraints assumed of the input, even though the period of the output is twice that of the input. In addition to the constraints captured by the geometry of the annulus, we add constraints as in [19] for the minimum time that  $\phi$  must remain in region 1 before entering region 2, and likewise for region 3. This construction allows a large class of input signals to be described in a simple and natural manner.

We specify the behavior of the toggle as a safety property. In particular, we use Coho to find an invariant subset of  $\mathbb{R}^d$  such that all trajectories in this set have a period twice that of the clock signal. This notion can be formalized using a Poincaré section [20]. Let  $\phi$  be the continuous signal corresponding to  $\Phi$ , and let  $c$  be some constant with  $V_{0h} < c < V_{1l}$ . Consider the intersection of the invariant set with the  $\phi = c$  hyperplane. These intersections form a Poincaré map [20]. We verify that these intersections form four disjoint regions (two for rising  $\phi$  crossing  $c$ , and two falling crossings). All trajectories must visit these four regions in the same order. Thus, the  $\phi = c$  plane partitions the invariant set of the continuous model into four, disjoint regions that map to the four discrete states of the discrete model.

Toggle elements can be composed to form a ripple-counter if there is an output variable such that for all trajectories in the manifold, the value and derivative of this variable satisfy the constraints of the input ring. It must also be shown that this output satisfies the minimum high and low time constraints. Section V shows that the toggle circuit satisfies all of these properties.

#### IV. MAKING COHO WORK

##### A. Solving LPs in Coho

Coho makes extensive use of linear programs (LPs), and the original version of Coho was hindered because these LPs are occasionally extremely ill-conditioned. The LPs in Coho have constraints that correspond to the convex hulls of projection polygons. These LPs are naturally written with the form

$$\min_x c^T x \text{ s.t. } Ax \leq b \quad (2)$$

where each row of  $A$  has at most two non-zero elements. We call such an LP a *Coho LP*. The dual of a Coho LP is a standard form LP:

$$\min_{\pi} -b^T \pi \text{ s.t. } A^T \pi = c \quad (3)$$

Note that  $A^T$  has at most two non-zero elements in any column. In [21], Laza presented a linear-time algorithm for solving the linear systems that arise when applying the Simplex algorithm to the dual of a Coho LP. With Laza's

algorithm, the tableau entries are computed from the original data at each pivot step; the only data carried forward from one pivot to the next is the original LP and the set of columns in the basis. By avoiding the rank-one updates of the tableau from the traditional formulation of Simplex, Laza's approach avoids error propagation from one pivot step to the next. His linear-time linear system solver makes this approach as efficient as traditional Simplex.

We implemented Laza's approach and included an arbitrary precision rational arithmetic package. Most computations are performed using interval arithmetic, and the arbitrary precision package is used for highly ill-conditioned bases and to verify the final solution. Our implementation has eliminated numerical stability problems from the LP solver. As an added benefit, our LP solver is guaranteed to find the exact optimum for the linear program in all cases. This guarantee allowed us to simplify many other parts of the Coho code.

##### B. Better Bloating

In the original formulation of Coho, the projection polygons were bloated equally in all directions to form the bloated convex hull that contained the trajectories for a time step. When verifying the toggle, we found that this resulted in large over approximations. Depending on the circuit state, some variables will have much larger time derivatives than others. If the state is bloated by the same amount for all variables, then the model will be evaluated with a much larger region than needed for the slow moving variables, and this causes the error term in the linear approximation of the model to be large.

We solved this problem by linearizing the models twice at each time step. The first phase is performed as in the original version of Coho, but we also keep track of the maximum magnitude of the derivative for each variable. At the end of the first phase, our new version computes the size of the allowed time-step. It also computes the bloat amount needed for each variable. In the second phase, faces are moved forward in time. In our new version, we linearize the model again for each face based on the per-variable bloat amounts computed from the first phase. This change enabled a dramatic reduction in the magnitude of the error terms in the linearizations of the model.

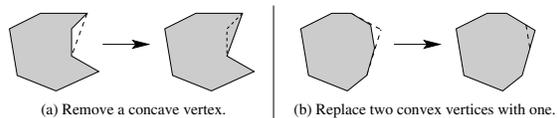


Fig. 5. Simplifying Projection Polygons

##### C. Simplifying Projection Polygons

At the end of each time step, Coho reduces the number of projection polygon vertices to keep their sizes tractable. As shown in Figure 5(a), Coho can simply delete convex vertices to produce an approximation of the original polygon.

Figure 5(b) shows the replacement of two consecutive, convex vertices with a single vertex, again producing an over approximation. Simplification serves two purposes. First it keeps the number of constraints in the LP for the bloated convex hull of the projectagon small. Second, it reduces the number of projectagon faces (i.e. projection polygon edges) that need to be advanced at each time step. These are separable concerns.

Our new version of Coho computes the convex hull of each polygon and simplifies it using the simplification shown in Figure 5(b) to control the number of constraints in the LP for the bloated convex hull,  $P_{bloat}$ . Then, we modified Coho to advance convex sequences of edges. Coho constructs LPs for these sequences and moves them forward in a single step. Of course, we still need to limit the growth of the number of polygon edges. The new Coho eliminates concave vertices as shown in Figure 5(a). Furthermore, once Coho identifies a convex sequence of vertices to advance, it can eliminate vertices as shown in Figure 5(b). Because we only have the extra edges for a single projection polygon at a time, we can maintain a much more detailed representation of the polygon without greatly increasing the time for the computation.

## V. VERIFYING THE TOGGLE

The verification that we present of the toggle resembles an earlier verification result presented in [19]. There are two significant differences in our approach and the earlier work. Most significantly, we include nodes *xx*, *yy* and *zz* in our model. This results in a seven-dimensional state space rather than the four-dimensional model used in [19]. Second, we model the drain-source currents of the transistors based on tabulated data obtained from HSPICE thus our results are based on the BSIM-3 models for the TSMC 180nm process. In [19], a simple, first-order, long-channel, MOSFET model was used for transistor current. Using a realistic model forced us to address other real-world issues, most notable of which was the leakage currents of the transistors. Thus, we added “keepers” to nodes *x*, *y* and *z*. While such practicalities can seem like a nuisance from a formal verification perspective, they show that our approach is solidly connected to the issues that challenge circuit designers for deep submicron processes.

We started our verification by simulating the toggle using a circuit simulation package that we have developed for use in Matlab. This approach had two advantages. First, we were able to informally validate our design before embarking on the more time-intensive effort of formal verification. Second, we modified the simulator to generate the linearized models with error bounds as required by Coho. Thus, we used the same circuit description for simulation and verification.

Our specification for the toggle requires it to have an invariant set that has twice the period of the input clock,  $\phi$ . Accordingly, our reachability calculation is carried out for two periods of  $\phi$ . We break each of these periods into the two phases: one for the rising transition of  $\phi$  and the time that  $\phi$  is high; and the other for the falling transition and low time. This partitioning had two advantages. First, we estimate a bounding hyper-rectangle for the end of each phase based

on the simulation results. With these estimates, we divide the task of verifying the toggle into four separate proof obligations where each obligation is of the form:

**Assume** that the circuit state is in hyper-rectangle  $Y_i$  at the end of phase  $i$ .

**Show** that the circuit state will be in hyper-rectangle  $Y_{i+1}$  at the end of phase  $i + 1$ .

By showing that the last phase leads to a hyper-rectangle that is contained in the initial hyper-rectangle of the initial phase, we establish that the reachable set that we have computed is invariant. Second, by separating the proof obligations, we can work on them in parallel. This was critical. Our current version of Coho is a prototype, and it is quite slow. It takes about a day to complete the reachability analysis for a single phase. Parallel computation allowed us to complete the verification, debug the model, and address issues of over approximation in a reasonably timely manner.

### A. Modeling the Circuit

We model transistors as voltage controlled current sources, and all capacitors as having fixed values with one terminal connected to ground. For transistor  $j$ , let  $ids_j(v_s, v_g, v_d)$  be the current that flows from the drain to the source when the voltages on the source, gate and drain are  $v_s$ ,  $v_g$  and  $v_d$  respectively. We obtained our  $ids$  functions by tabulating data on a 0.01 volt grid for  $(v_s, v_g, v_d) \in [-0.3, 2.25]^3$  for transistors in the TSMC 180nm, 1.8 volt, bulk CMOS process. We ignore gate leakage which is negligible in the 180nm process.

By Kirchoff’s current law, the total current flowing out of each node through the capacitors connected to the node must equal the total current flowing into the node through the transistors. The current through a capacitor is  $c\dot{v}$  where  $c$  is the capacitance of the capacitor and  $\dot{v}$  is the time derivative of the voltage across the capacitor. This yields:

$$\dot{V} = C^{-1} M ids(V) \quad (4)$$

where  $V$  is the vector of node voltages (one element for each node of the circuit);  $ids(V)$  is the vector of drain-to-source currents (one element for each transistor);  $M$  maps transistors to nodes with  $M(i, j) = 1$  if the source of transistor  $j$  is connected to node  $i$ ,  $M(i, j) = -1$  if the drain of transistor  $j$  is connected to node  $i$ , and  $M(i, j) = 0$  otherwise.  $C$  is the matrix of inter-node capacitances. Because we model all capacitances as being fixed and to ground,  $C$  is fixed and diagonal.

Equation 4 is the basis for our circuit modeling. To create a linear model as in Equation 1, it suffices to linearize  $ids$ . Here, we’ll present our initial approach. Given a linear program that contains the bloated face for which Coho needs a model, we obtain upper and lower bounds for  $v_s$ ,  $v_g$  and  $v_d$ . We then compute a linear regression on the points in this bounding box using tables of precomputed sums. Noting that the  $ids$  function has exactly one inflection, (along the  $v_s = v_d$  plane) enables efficient computation of the worst-case errors of the least-squares model. We then adjust the constant term from the

linear-regression to balance the positive and negative worst-case errors. We now have

$$\begin{aligned} & (C^{-1} M A_{ids})V + (C^{-1} M A_{ids})b_{ids} - (C^{-1} M A_{ids})u_{ids} \\ \leq & \hat{V} \\ \leq & (C^{-1} M A_{ids})V + (C^{-1} M A_{ids})b_{ids} + (C^{-1} M A_{ids})u_{ids} \end{aligned} \quad (5)$$

where  $A_{ids}$  is computed from the linear regression and  $b_{ids}$  and  $u_{ids}$  are determined by the error analysis.

This model is simple, efficient to compute and worked fairly well for the toggle example. Section V-B describes some specific situations where the error term from this model was too large to complete the verification of the toggle and how we refined the model to achieve a successful verification.

Finally, we need a model for the input signal,  $\phi$ , which is specified as satisfying a Brockett annulus. We obtained a candidate annulus by simulating the toggle with inputs of varying amplitude and frequency and observing the Brockett annulus satisfied by the toggle’s output. We described this annulus by giving polygons for the inner and outer rings. Figure 6 shows the annulus that we used. We set the minimum low and high times for  $\phi$  to 0.5ns. The circuit model uses the LP for the current bloated face to obtain bounds on  $\phi$ . The model computes the center path for  $\phi$  in this interval, computes the minimum, least-squares approximation of this path, and finds the worst-case errors for this approximation.

### B. The Reachability Computation

As described above, we divided the reachability computation into four phases according to the four state transitions shown in Figure 3. Table I summarizes this analysis. We start each phase with the projectagon for the starting hyper-rectangle and note the bounding hyper-rectangle of the projectagon for the reachable region at the end of each phase. Note that the starting hyper-rectangle for each phase contains the ending hyper-rectangle of the previous phase, and the starting hyper-rectangle for the phase 1 contains the ending hyper-rectangle for phase 4. Thus, we’ve established an invariant set. Furthermore, the hyperrectangles for the four phases are pairwise disjoint. Thus, this invariant set has a period of two with respect to the clock input,  $\phi$ .

Table I also lists the projection polygons that we used for each phase. These were chosen with two considerations. First, we chose projections that corresponded to logical dependencies between changing signals. Thus, in the first phase when  $z$  changes, we include  $z$  vs.  $zz$  and  $z$  vs.  $x$  (because the falling edge of  $z$  enables a rising edge of  $x$ ). Second, we included at least one polygon for each variable to bound the resulting projectagon in all dimensions.

We now describe the verification of each of these phases in greater detail.

*Phase 1:* ( $\phi = 0, x = 0, y = 1, z = 1$ )  $\rightarrow$  ( $\phi = 1, x = 0, y = 1, z = 0$ ): In this phase,  $\phi$  makes a low-to-high transition, and  $z$  goes from high-to-low. This phase starts with  $\phi = 0.2$  volts at which point  $\phi$  is already in region 2 of the Brockett annulus. This ensures that  $\dot{\phi}$  is strictly positive. This phase includes the rising transition of  $\phi$  (region 2 of the

annulus) and the time that  $\phi$  is high (region 3). We compute the union of the reachable regions for all times starting once  $\phi$  has been high for the minimum required time. When this union computation reaches a fixpoint, we have the reachable set for the end of this phase. In the next phase,  $\phi$  will enter region 4 of the Brockett annulus. We over approximate the annulus by lowering the minimum high value for  $\phi$  to 1.6 volts (from 1.65) during this phase of the verification. This allows us to use a starting value of 1.6 volts for  $\phi$  in the next phase which will ensure that  $\dot{\phi}$  is strictly negative.

The linear model for  $\phi$  has large errors if the interval for  $\phi$  is too large. Thus, we “sliced” the space into regions corresponding to 0.1 volt wide intervals for  $\phi$ . Consider the scenario when Coho is working on a slice with  $\phi \in [lo, hi]$  and let  $P_0$  be the projectagon at the end of a time step. Coho divides the  $P_0$  into a portion,  $P_{lo}$ , with  $\phi \leq hi$  and a portion,  $P_{hi}$  with  $\phi \geq lo$ . To obtain  $P_{lo}$ , Coho intersects each projection polygon that includes  $\phi$  in its basis with the  $\phi \leq hi$  half-plane. Projection polygons in other bases are left unchanged. This ensures that  $P_{lo} \subseteq P_0 \cap \{u \mid u_\phi \leq lo\}$ . Coho computes  $P_{hi}$  for the timestep in the same manner. Because  $\dot{\phi} > 0$  when  $\phi$  is in region 2 of the Brockett ring, Coho will eventually reach a time step when  $P_{lo}$  is empty.

For each projection polygon basis, Coho computes the union of the polygons that it computed for this basis for  $P_{hi}$  to obtain the projection polygon that it will use for that basis in the next slice of  $\phi$ . It is straightforward to show that the union of two projectagons is contained in the projectagon obtained from the union of their projection polygons. Occasionally, the polygons for two consecutive time steps will be disjoint – this can occur when the projectagon barely crosses into the  $\phi \geq hi$  half-space. In this case, Coho uses the bounding box of the union as a simple, over approximation.

The transistor model from Section V-A can produce large error bounds that include currents that flow against the drain-to-source voltage. These non-physical behaviours allowed by the model caused Coho to fail to verify the toggle. It is simple to show that the circuit model has an invariant that all node voltages are between 0 volts (i.e. ground) and 1.8 volts (i.e.  $V_{dd}$ ) and that  $xx \geq x, yy \leq y$  and  $zz \leq z$ . We modified Coho to allow the user-supplied model to provide such invariants. With these user-supplied invariants, Coho successfully computed the tight bounds on the reachable region at the end of the phase described above.

*Phase 2:* ( $\phi = 1, x = 0, y = 1, z = 0$ )  $\rightarrow$  ( $\phi = 0, x = 1, y = 1, z = 0$ ): In this phase,  $\phi$  makes a high-to-low transition, and  $x$  goes from low-to-high. The verification proceeded in the same manner as for Phase 1.

*Phase 3:* ( $\phi = 0, x = 1, y = 1, z = 0$ )  $\rightarrow$  ( $\phi = 1, x = 0, y = 0, z = 1$ ): In this phase,  $\phi$  goes from low-to-high, and all three of  $x, y,$  and  $z$  change their values, in the order  $y \downarrow \rightarrow z \uparrow \rightarrow x \downarrow$ . This was the most challenging phase to verify. As seen in Table I, we used ten projection polygons for this phase instead of six as were used in the other phases.

The greatest challenge arose because  $z$  can start its rising transition while  $y$  is still falling. To show that output,  $q$  of the

Start and end hyper-rectangle for each phase							
Phase	$\phi$	$x$	$y$	$z$	$xx$	$yy$	$zz$
1, start	0.2	[0.000, 0.100]	[1.700, 1.800]	[1.700, 1.800]	[0.000, 1.0]	[0.000, 0.100]	[0.000, 0.100]
1, end	1.6	[0.000, 0.002]	[1.790, 1.800]	[0.000, 0.014]	[1.788, 1.8]	[0.000, 0.004]	[0.000, 0.001]
2, start	1.6	[0.000, 0.100]	[1.700, 1.800]	[0.000, 0.100]	[1.700, 1.8]	[0.000, 0.100]	[0.000, 0.100]
2, end	0.2	[1.795, 1.800]	[1.758, 1.800]	[0.000, 0.043]	[1.795, 1.8]	[1.152, 1.736]	[0.000, 0.003]
3, start	0.2	[1.750, 1.800]	[1.750, 1.800]	[0.000, 0.050]	[1.750, 1.8]	[0.800, 1.800]	[0.000, 0.040]
3, end	1.6	[0.000, 0.001]	[0.000, 0.005]	[1.703, 1.800]	[1.785, 1.8]	[0.000, 0.002]	[0.843, 1.740]
4, start	1.6	[0.000, 0.100]	[0.000, 0.100]	[1.700, 1.800]	[1.700, 1.8]	[0.000, 0.100]	[0.800, 1.800]
4, end	0.2	[0.000, 0.100]	[1.700, 1.800]	[1.700, 1.800]	[0.000, 1.0]	[0.000, 0.100]	[0.000, 0.100]

Projection Polygons for each phase	
Phase	Polygons
1	$x$ vs. $xx$ , $x$ vs. $z$ , $z$ vs. $zz$ , $z$ vs. $xx$ , $\phi$ vs. $y$ , $\phi$ vs. $yy$
2	$x$ vs. $xx$ , $x$ vs. $y$ , $x$ vs. $yy$ , $y$ vs. $yy$ , $\phi$ vs. $z$ , $\phi$ vs. $zz$
3	$x$ vs. $xx$ , $x$ vs. $y$ , $x$ vs. $yy$ , $y$ vs. $yy$ , $y$ vs. $z$ , $y$ vs. $zz$ , $z$ vs. $zz$ , $z$ vs. $z$ , $z$ vs. $xx$ , $\phi$ vs. $z$
4	$x$ vs. $xx$ , $y$ vs. $yy$ , $y$ vs. $z$ , $y$ vs. $zz$ , $z$ vs. $zz$ , $\phi$ vs. $z$

TABLE I  
REACHABILITY SUMMARY

toggle satisfies the same Brockett annulus as used for  $\phi$  (see Section V-C), the transitions of  $z$  need to have relatively small rise and fall times. The time derivative of  $z$  depends on the values of  $\phi$ ,  $y$ ,  $z$  and  $zz$ . We found that once  $\phi$  was high (i.e. greater than 1.6 volts), it was helpful to slice on the value of  $y$ . We used 0.1 volt wide slides for  $y$  as it fell from 1.3 volts to 0.1 volts. We sliced  $z$  in the same manner but found that it was unnecessary to slice  $x$ .

For  $z$  to make its rising transition,  $zz$  must also rise. Otherwise, the current through the transistor between  $z$  and  $zz$  can dominate the current through the pull-up transistor for  $z$ . We found it surprisingly challenging to show that  $zz$  rises at an acceptable rate. The problem occurs when both  $z$  and  $zz$  have wide bounds including values close to ground for each. With wide bounds, the error term for the current through the transistor between  $z$  and  $zz$  can be large. At one extreme of this error bound, the current from  $z$  to  $zz$  is negative which keeps the lower bound for  $zz$  at ground. At the other extreme, the current from  $z$  to  $zz$  is large and positive which keeps  $z$  from rising.

We solved the negative current problem by adding a transistor model that simply determines the minimum and maximum drain-to-source current for the region around the current face. While this model has a large error-term, it never predicts a current of the wrong sign. We modified Coho to each compute forward time step twice: first using the least-squares current models and then using the min/max models. Coho then computes the intersection of the two projectagons that it obtains. Because each of the projectagons contains the actual reachable space, their intersection does as well. This preserves the soundness of Coho while significantly reducing the errors.

Finally, we introduced the improvements in the bloat calculation described in Section IV-B and polygon simplification described in Section IV-C. With these changes, we were able to verify this phase (and all four phases) of the toggle's operation.

*Phase 4:* ( $\phi = 1, x = 0, y = 0, z = 1$ )  $\rightarrow$  ( $\phi = 0, x = 0, y = 1, z = 1$ ): In this phase,  $\phi$  makes a high-to-

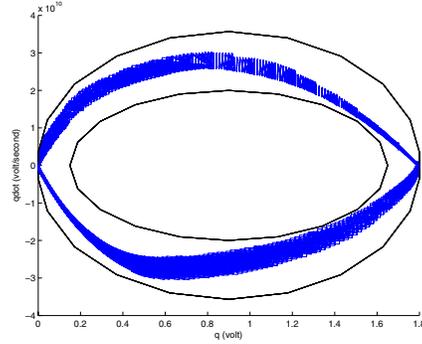


Fig. 6. The Brockett Annulus for  $q$  and  $\dot{q}$

low transition, and  $y$  goes from low-to-high. The verification proceeded in the same manner as for Phases 1 and 2.

### C. Verifying the Output Brockett Annulus

Thus far, we have ignored the  $q$  output of the toggle in our analysis – we simply included a load on  $z$  equal to the gate capacitance of transistors that drive  $q$ . We verified the operation of the inverter separately. To do so, we first constructed the Brockett annulus for the  $z$  output.

At each time step of the verification described above, we determined the reachable combinations of  $z$  and  $\dot{z}$ . We note that  $\dot{z}$  is negative monotonic in  $z$  and positive monotonic in  $zz$ . Thus, the extremal values of  $z$  vs.  $\dot{z}$  occur on the boundary of the  $z$  vs.  $zz$  projection. For each edge of the  $z$  vs.  $zz$  projection, Coho computes the linearized circuit model and uses this model to find the reachable combinations of  $z$  and  $\dot{z}$ . From these, we constructed a Brockett annulus that is satisfied by  $z$  and  $\dot{z}$ .

We then perform a separate reachability analysis for the output inverter. The input to this circuit is modeled by the Brockett annulus derived above, and we compute the reachable

region of  $q$  vs.  $\dot{q}$  as described above for  $z$ . Figure 6 shows the result;  $q$  clearly satisfies the constraints that we used for  $\phi$ . Thus, these toggles can be composed to form an arbitrarily large ripple-counter as desired.

## VI. CONCLUSIONS

We have implemented a working version of the Coho algorithm for performing reachability analysis of circuits modeled by non-linear differential equations. We used Coho to verify a toggle flip-flop using a model that exposed all seven nodes of the circuit and used accurate (vendor provided BSIM-3) transistor models. We believe that our verification of a seven-dimensional, non-linear system is the largest such verification to date and shows the feasibility of using formal methods to verify digital designs at the circuit level.

Coho is slow, and our immediate work will be to address issues of performance. We are aware of obvious opportunities for improving the performance including more efficient algorithms for some of the critical operations, and re-implementing critical pieces of code in C instead of the current Matlab and Java implementation. Furthermore, there is abundant parallelism available in Coho, as each edge of each projection polygon can be processed independently during each time step. The multithreading capabilities of the Java part of the code should make this parallelism readily accessible and we will explore parallel implementations. We note that the memory requirements for Coho are relatively modest, and unlike many reachability tools, we don't expect memory to be a critical limitation in the near future.

In the process of verifying the toggle, we have identified numerous places where the over approximations of our models and algorithms could be reduced at a relatively low computational cost. We will explore these refinements. This should enable the verification of more complex circuits. While climbing up the ladder to higher-dimensional models is challenging, we note that it is also very rewarding. The fraction of interesting cell designs that can be modeled grows rapidly with dimension, and we expect that a tool that could verify circuits with ten to fifteen nodes would be adequate for most cells in a cell-library or specialized logic functions in a full-custom design. Once cells are verified with specifications that allow compositional reasoning, more traditional tools for discrete equivalence and model checking could be used to verify larger designs.

This first demonstration of verifying a circuit with a detailed, non-linear circuit model was far from automatic. Significant human expertise was required in using Coho, improving the reachability algorithms, modeling the circuits, and writing the specification. This is typical for the first demonstration of a new verification technology. We expect that as we improve Coho and apply it to a wider range of examples, we will also develop a more automated verification flow that will allow these verification techniques to be used by typical circuit designers.

## REFERENCES

- [1] J. Yuan and C. Svensson, "High-speed CMOS circuit technique," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 62–70, Feb. 1989.
- [2] M. R. Greenstreet and I. Mitchell, "Integrating projections," in *Proceedings of the First International Workshop on Hybrid Systems: Computation and Control*, T. A. Henzinger and S. Sastry, Eds., Berkeley, California, Apr. 1998, pp. 159–174.
- [3] —, "Reachability analysis using polygonal projections," in *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*. Berg en Dal, The Netherlands: Springer, Mar. 1999, pp. 103–116, LNCS 1569.
- [4] R. Alur, T. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 181–201, 1996.
- [5] T. Henzinger, J. Preussig, and H. Wong-Toi, "Some lessons from the HyTech experience," in *Proceedings of the 40th Annual Conference on Decision and Control*. IEEE Press, 2001, pp. 2887–2892.
- [6] T. A. Henzinger, "The theory of hybrid automata," in *Proceeding of the 11<sup>th</sup> Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, 1996, pp. 278–292.
- [7] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Algorithmic analysis of nonlinear hybrid systems," *IEEE Transactions on Automatic Control*, vol. 43, no. 4, pp. 540–554, Apr. 1998.
- [8] E. Asarin and O. Maler, "On the analysis of dynamical systems having piecewise-constant derivatives," *Theoretical Computer Science*, vol. 138, pp. 35–65, 1995.
- [9] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, pp. 94–124, 1999.
- [10] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Proceedings of the Fifth International Workshop on Hybrid Systems: Computation and Control*. Springer-Verlag, 2005, pp. 258–273, LNCS 3414.
- [11] R. Bagnara *et al.*, "Possibly not closed convex polyhedra and the Parma polyhedra library," in *Proceedings of the International Symposium on Static Analysis*, 2002, pp. 213–229, LNCS 2477.
- [12] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *Proceedings of Design Automation and Test Europe*, Mar. 2006, pp. 257–262.
- [13] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *Proceedings of the Fourteenth Conference on Computer Aided Verification*. Copenhagen: Springer, July 2002, pp. 365–370.
- [14] O. Bournez, O. Maler, and A. Pnueli, "Orthogonal polyhedra: Representation and computation," in *Proceedings of the Second International Workshop on Hybrid Systems: Computation and Control*. Springer, 1999, pp. 46–60, LNCS 1569.
- [15] R. Kurshan and K. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 11, pp. 1356–1371, Nov. 1991.
- [16] B. Silva, K. Richeson, *et al.*, "Modeling and verifying hybrid dynamical systems using *checkmate*," in *Proceedings of the 4<sup>th</sup> International Conference on Automation of Mixed Processes (ADPM 2000)*, 2000, pp. 323–328.
- [17] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *Proceedings of 2004 IEEE/ACM International Conference on Computer Aided Design*, Nov. 2004, pp. 210–217.
- [18] R. Brockett, "Smooth dynamical systems which realize arithmetical and logical operations," in *Three Decades of Mathematical Systems Theory: A Collection of Surveys at the Occasion of the 50th Birthday of J. C. Willems*, ser. Lecture Notes in Control and Information Sciences, H. Nijmeijer and J. M. Schumacher, Eds. Springer, 1989, vol. 135, pp. 19–30.
- [19] M. R. Greenstreet, "Verifying safety properties of differential equations," in *Proceedings of the 1996 Conference on Computer Aided Verification*, New Brunswick, NJ, July 1996, pp. 277–287.
- [20] T. S. Parker and L. O. Chua, *Practical Numerical Algorithms for Chaotic Systems*. New York: Springer, 1989.
- [21] M. D. Laza, "A robust linear program solver for projectahedra," Master's thesis, Department of Computer Science, University of British Columbia, Vancouver, BC, Dec. 2001.

# Combining Symbolic Simulation and Interval Arithmetic for the Verification of AMS Designs

Mohamed H. Zaki\*, Ghiath Al-Sammane\*, Sofiène Tahar\*, and Guy Bois ‡

\*Electrical & Computer Engineering

Concordia University, Montreal, Quebec, Canada

Email: {mzaki, sammane, tahar}@ece.concordia.ca

‡Genie Informatique

Ecole Polytechnique de Montreal, Montreal, Quebec, Canada

Email: guy.bois@polymtl.ca

**Abstract**—Analog and mixed signal (AMS) designs are important integrated circuits that are usually needed at the interface between the electronic system and the real world. Recently, several formal techniques have been introduced for AMS verification. In this paper, we propose a difference equations based bounded model checking approach for AMS systems. We define model checking using a combined system of difference equations for both the analog and digital parts, where the state space exploration algorithm is handled with Taylor approximations over interval domains. We illustrate our approach on the verification of several AMS designs including  $\Delta\Sigma$  modulator and oscillator circuits.

## I. INTRODUCTION

Analog and mixed signal (AMS) designs are important integrated circuits used at the interface between an electronic system and its real world. Several computer aided design tools for AMS systems have been developed to overcome challenges in the design process of such designs. Simulation based verification approaches are usually applied to check that an AMS design is robust with respect to different types of inaccuracies. However, with circuits growing in complexity, simulation is not enough to validate complex properties. Actually, it is reported that in recent chips about 50% of errors that implied redesign are due to errors in analog or mixed portions [19]. Therefore, introducing new verification methodologies for these systems is growing in importance.

Boosted by previous successes in the verification of corner cases in digital designs, formal methods became a serious candidate for the verification of AMS systems. In fact, they promise a complete verification and a high level of confidence. Usually, one is interested in global properties connected to the dynamic behavior of the AMS systems. For example, we might be interested in reachability properties, like “can we reach from the initial state a state where a certain condition holds?” or “will the circuit oscillate for giving parameters?”. Unfortunately, a direct application of formal methods on AMS systems is very difficult. Unlike digital designs, the functionality of AMS systems is defined in terms of continuous quantities and in terms of continuous time, as they deal usually with factors like voltage level, signal noise and current leakage, in addition to higher order physical effects when designing in deep submicron. In fact, while the behavior of AMS systems is

generally modeled using differential equations over continuous quantities, formal methods, however, are defined using models based on discrete events and automata. Today, an important gap remains in linking these two mathematical approaches.

Most research efforts concentrate on how to abstract differential equations in order to be adopted inside automata based algorithms. In this paper, we propose an alternative approach, based on bounded model checking [5] for AMS systems modeled in terms of recurrence equations. Discrete and continuous time based analog systems are described using ordinary differential equations or difference equations, respectively, while the digital parts of the AMS design are described using event based models. We then define a model checking method using a combined system of difference equations for both the analog and digital parts, where state space exploration algorithms are handled with Taylor approximations over interval domains. Such modeling allows the computation over continuous quantities while avoiding the unsoundness inherent in the numerical Taylor approximation. We illustrate the proposed method on the verification of a variety of designs including a  $\Delta\Sigma$  modulator design and oscillator circuits.

The rest of the paper is organized as follows: In Section II, we give an overview of the proposed methodology, followed by system model description in Section III. Interval based analysis and Taylor models are described in Section IV. The verification algorithm along with symbolic simulation are then presented in Section V. Experimental results are shown in Section VI, and finally, in Section VII, we present related work before we conclude the paper with Section VIII.

## II. PROPOSED METHODOLOGY

The principle of bounded model checking (BMC) is the search for a counter-example of the property checked against the model for a bounded  $k$  steps. If such counter-example is found or a fixpoint is reached, the verification task is achieved, else the number of steps can be increased for further verification.

An AMS system is a hybrid system composed in general of a digital part described using logical primitives and an analog part which can be described directly using recurrence

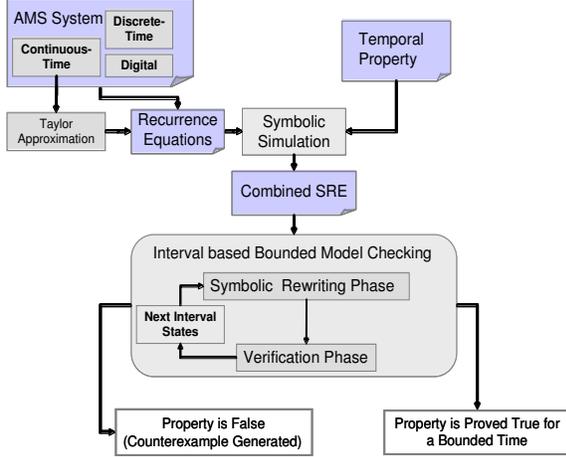


Fig. 1. Overview of the AMS Verification Methodology

equations or a set of differential equations. We propose to convert differential equations into an equivalent set of recurrence equations using the Taylor approximation method. Therefore the recurrence model gives the possibility to handle continuous behaviors like that of current and voltages, but in discrete time intervals, which cover a non-trivial class of mixed behaviors. The properties are temporal relations between signals of the system and are described using a basic subset of Linear Temporal Logic (LTL).

The proposed methodology is composed of two steps as shown in Figure 1. In the first step, the AMS description and LTL property of interest are input to a symbolic simulator that performs a set of transformations by rewriting rules in order to obtain a mathematical representation called System of Generalized Recurrence Equations (SRE) (to be described later). These are recurrence relations that give a description of the property of interest in terms of the system equations. The next step is to prove the properties using a verification engine that performs bounded model checking over interval Taylor model forms. The interval Taylor model form is a combined symbolic numerical representation of the system equations using polynomials and interval terms that ensure enclosure of the reachable states, hence providing a sound abstraction of the reachable sets.

We have implemented this verification algorithm using the computer algebra system *Mathematica*, which provides special functions for symbolic simplification, manipulation and proof of algebraic relations.

### III. AMS DESIGN MODELING

Different formalisms have been proposed for modeling systems with combination of discrete and continuous (hybrid) behavior, for instance, hybrid automata [14]. Such formalisms have been applied for AMS modeling. In this paper, we propose to use a generalization of recurrence equations to model different aspects of the AMS designs; mainly the continuous and discrete time behaviors.

#### A. Systems of Recurrence Equations

The notion of recurrence equation was extended in [1] to describe digital circuits using what is called generalized If-formula. Such formalization was found practical in modeling hybrid systems like discrete-time AMS design [2]. In the remaining of this paper, we will show how such recurrence equations can be suitable under certain conditions for modeling continuous-time AMS systems, hence allowing a unified modeling framework for discrete and continuous time AMS designs.

##### Definition 1: Generalized If-formula

In the context of symbolic expressions, the generalized If-formula is a class of expressions that extend recurrence equations to describe digital systems. Let  $\mathbb{K}$  be a numerical domain ( $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$  or  $\mathbb{B}$ ), a generalized If-formula is one of the following:

- A variable  $x_i(n) \in \mathbf{x}(n)$ , with  $i \in \{1, \dots, d\}$ ,  $n \in \mathbb{N}$  and  $\mathbf{x}(n) = \{x_1(n), \dots, x_d(n)\}$ .
- A constant  $C \in \mathbb{K}$
- Any arithmetic operation  $\diamond \in \{+, -, \div, \times\}$  between variables  $x_i(n) \in \mathbb{K}$
- A logical formula: any expression constructed using a set of variables  $x_i(n) \in \mathbb{B}$  and logical operators: *not, and, or, xor, nor, ...*, etc.
- A comparison formula: any expression constructed using a set of  $x_i(n) \in \mathbb{K}$  and comparison operator  $\alpha \in \{=, \neq, <, \leq, >, \geq\}$ .
- An expression  $IF(X, Y, Z)$ , where  $X$  is a logical formula or a comparison formula and  $Y, Z$  are any generalized If-formula. Here,  $IF(x, y, z) : \mathbb{B} \times \mathbb{K} \times \mathbb{K} \rightarrow \mathbb{K}$  satisfies the axioms:
  - (1)  $IF(True, X, Y) = X$
  - (2)  $IF(False, X, Y) = Y$

##### Definition 2: A System of Recurrence Equations (SRE)

Consider a set of variables  $x_i(n) \in \mathbb{K}$ ,  $i \in \{1, \dots, d\}$ ,  $n \in \mathbb{N}$ , an SRE is a system consisting of a set of equations of the form:

$$x_i(n) = f_i(x_j(n - \gamma)), (j, \gamma) \in \varepsilon_i, \forall n \in \mathbb{Z}$$

where  $f_i(x_j(n - \gamma))$  is a generalized If-formula. The set  $\varepsilon_i$  is a finite non-empty subset of  $1, \dots, d \times \mathbb{N}$ . The integer  $\gamma$  is called the delay.

*Example 1:* Figure 2 shows a first-order  $\Delta\Sigma$  modulator of one-bit with two quantization levels, +1V and -1V. Consider the constraint that the quantizer (input signal  $y(n)$ ) should be between -2V and +2V in order to not be overloaded. The SRE of the  $\Delta\Sigma$  is then described as:

$$\begin{aligned} y(n) &= y(n-1) + u(n) - v(n-1) \\ v(n-1) &= IF(y(n-1) > 0, 1, -1) \end{aligned}$$

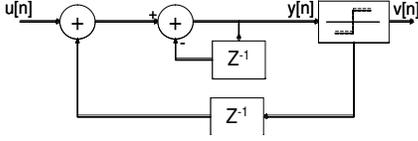


Fig. 2. First-order  $\Delta\Sigma$  Modulator

### B. Taylor Approximation

A large class of AMS and analog designs have continuous time behavior, usually described using a system of ordinary differential equations (ODE). Unfortunately, a closed form solution is generally not available for ODE systems and discrete approximate models are used. One basic idea is to use the approximation  $\mathbf{x}[t_{k+1}] = f(\mathbf{x}[t_k]) + \mathcal{R}m$  of the ODE  $\dot{\mathbf{x}} = f(\mathbf{x})$  as truncated Taylor series for  $\mathbf{x}(t)$ , expanded about time instant  $t_k$ , with a remainder term  $\mathcal{R}m$ .

**Theorem 1: Taylor Approximation.** Suppose a function  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  over states vector  $\mathbf{x} \in \mathbb{R}^d$  is  $m + 1$  time partially differentiable on the interval  $[a, b]$ . Assume  $\mathbf{x}_0 \in [a, b]$ , such that  $a, b \in \mathbb{R}^d$ , then for each  $\mathbf{x} \in [a, b]$ ,  $\exists \lambda \in \mathbb{R}$ ,  $0 \leq \lambda \leq 1$ , such that:

$$f(\mathbf{x}) = \sum_{k=0}^m \frac{[(\mathbf{x} - \mathbf{x}_0) \cdot \nabla]^k f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0}}{k!} + \frac{[(\mathbf{x} - \mathbf{x}_0) \cdot \nabla]^{m+1} f(\mathbf{x})|_{\mathbf{x}=\Lambda}}{(m+1)!}$$

where  $\nabla = \mathbf{i}_1 \frac{\partial}{\partial x_1} + \dots + \mathbf{i}_d \frac{\partial}{\partial x_d}$  and  $\Lambda = \mathbf{x}_0 + \lambda(\mathbf{x} - \mathbf{x}_0)$

In general, to obtain an approximate solution of the ODE system, we consider a sequence of discrete time points  $t_0, t_1, \dots, t_m$  for which the solution is approximated, with  $h_i = t_{i+1} - t_i$ . If the solution  $\mathbf{x}(t)$  of an ODE system  $\dot{\mathbf{x}} = f(\mathbf{x})$  is a function which is  $p + 1$  times continuously differentiable on the open interval  $(t_i, t_{i+1})$ , then, from the Taylor approximation theorem, we have:

$$\mathbf{x}(t_{i+1}) = \mathbf{x}(t_i) + \sum_{k=1}^p \left( \frac{h^k}{k!} \mathbf{x}^{(k)}(t_i) \right) + \left( \frac{h^{p+1}}{(p+1)!} \mathbf{x}^{(p+1)}(\xi) \right)$$

with  $h = t_{i+1} - t_i$  and  $\xi = [t_i, t_{i+1}]$  and  $\forall k \in [1, p+1]$ ,  $\mathbf{x}^{(k)} = f^{(k-1)}(\mathbf{x}(t), t)$ , where the vector function  $f$  is composed by  $d$  elementary functions  $f_q(x_1, \dots, x_d)$ ,  $q \in \{1, \dots, d\}$ , such that:

$$f_q^{(k)}(x_1, \dots, x_d) = \sum_{m=1}^d \left( \frac{\partial^k f_q}{\partial x_m} (x_1, \dots, x_d) \right) f_m(x_1, \dots, x_d)$$

Such representation allows giving an approximate polynomial description of the behavior of an ODE system using generalized SRE. To preserve the inherited behavior of the actual solution, the remainder term should not be discarded and instead bounds must be specified. We use interval arithmetic methods to obtain such bounds. Interval arithmetic provides an over-approximation of the original

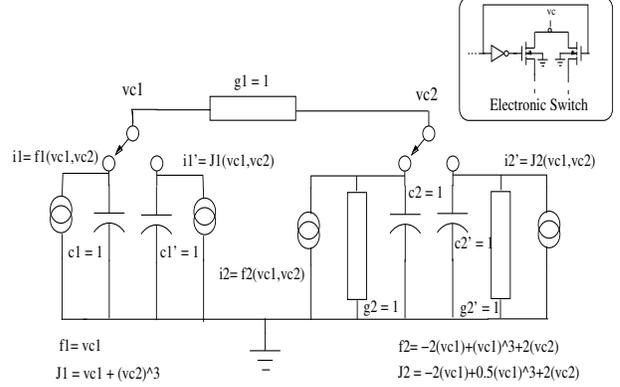


Fig. 3. Switched Analog Circuit

behavior of the system as will be shown later on.

**Example 2:** Consider the analog circuit in Figure 3, composed of a network of passive components (capacitors and conductances), along with non-linear current sources and two switches. The switches can be designed using CMOS transistors working in saturation mode as shown in the figure. This circuit exhibits an oscillatory behavior when the initial capacitor voltages are within a specified range, based on the switches positions. The voltages across the capacitors can be described using ODEs as follows:

$$\begin{cases} v_{c1} = v_{c2} & \text{or} & v_{c1} = v_{c2} + v_{c2}^3 \\ v_{c2} = -v_{c1} + v_{c1}^3 & \text{or} & v_{c2} = -v_{c1} + (1/2)v_{c1}^3 \end{cases}$$

Suppose that we specify the switching conditions as

$$Cond_1 = Cond_2 := v_{c1}(n-1) \leq v_{c2}(n-1)$$

For illustration purposes and for clarity, we use Taylor approximation limited to order 2 to obtain the corresponding SREs:

$$v_{c1}(n) := IF(Cond_1, X_1, X_2) \quad \text{and} \quad v_{c2}(n) := IF(Cond_2, Y_1, Y_2)$$

with:

- $X_1 := \frac{h^2 v_{c1}(n-1)^3}{2} - \frac{h^2 v_{c1}(n-1)}{2} + v_{c1}(n-1) + h v_{c2}(n-1) + \mathcal{R}m_1[\widetilde{v}_{c1}, \widetilde{v}_{c2}]$
- $X_2 := \frac{h^2 v_{c1}(n-1)^3}{4} + \frac{3}{4} h^2 v_{c2}(n-1)^2 v_{c1}(n-1)^3 - \frac{h^2 v_{c1}(n-1)}{2} - \frac{3}{2} h^2 v_{c2}(n-1)^2 v_{c1}(n-1) + v_{c1}(n-1) + h v_{c2}(n-1)^3 + h v_{c2}(n-1) + \mathcal{R}m_2[\widetilde{v}_{c1}, \widetilde{v}_{c2}]$
- $Y_1 := h v_{c1}(n-1)^3 + \frac{3}{2} h^2 v_{c2}(n-1) v_{c1}(n-1)^2 - h v_{c1}(n-1) - \frac{h^2 v_{c2}(n-1)}{2} + v_{c2}(n-1) + \mathcal{R}m_3[\widetilde{v}_{c1}, \widetilde{v}_{c2}]$
- $Y_2 := \frac{h v_{c1}(n-1)^3}{2} + \frac{3}{4} h^2 v_{c2}(n-1)^3 v_{c1}(n-1)^2 + \frac{3}{4} h^2 v_{c2}(n-1) v_{c1}(n-1)^2 - h v_{c1}(n-1) - \frac{h^2 v_{c2}(n-1)^3}{2} - \frac{h^2 v_{c2}(n-1)}{2} + v_{c2}(n-1) + \mathcal{R}m_4[\widetilde{v}_{c1}, \widetilde{v}_{c2}]$

where  $\mathcal{R}m_i[\widetilde{v}_{c1}, \widetilde{v}_{c2}]$  are the Taylor approximation remainders,  $i = \{1, \dots, 4\}$  and  $h$  is the time step.

#### IV. INTERVAL ANALYSIS

Interval domains are numerical domains that enclose the original states of a system of equations at each discrete step [20]. Algorithms supporting such numerical domains are used to produce bounded envelopes for the reachable states not only at some discrete time points but also for all continuous ranges of intermediate states between any two consecutive time discrete points. These algorithms, generally known as *validated methods*, are an attractive tool to use in the verification of the behavior of systems with uncertainty on the design parameters or initial conditions. The fact that the generated bounds provide a sound abstraction for the reachable states, makes it attractive to be used with formal verification techniques. The basic interval arithmetics is defined as follows:

Let  $I_1 = [a, b]$  and  $I_2 = [a', b']$  be two real intervals (bounded and closed), the basic arithmetic operations on intervals are defined by:

$$I_1 \Phi I_2 = \{r_1 \Phi r_2 \mid r_1 \in I_1 \wedge r_2 \in I_2\}$$

with  $\Phi \in \{+, -, \times, /\}$  except that  $I_1/I_2$  is not defined if  $0 \in I_2$  [20].

In addition, other elementary functions can be included as basic interval arithmetic operators. For example, *exp* may be defined as  $\text{exp}([a, b]) = [\text{exp}(a), \text{exp}(b)]$ .

The guarantee that the real solutions for a given function are enclosed by the interval representation is formalized by the following property.

**Definition 3: Inclusion Function.**[20] Let  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  be a continuous function, then  $F : \mathbb{I}^d \rightarrow \mathbb{I}$  is an interval extension (inclusion function) of  $f$  if

$$\{f(x_1, \dots, x_d) \mid x_1 \in X_1, \dots, x_d \in X_d\} \subseteq F(X_1, \dots, X_d)$$

where  $\mathbb{I}$  is the interval domain and  $X_i \in \mathbb{I}$ ,  $i \in \{1, \dots, d\}$ .

Inclusion functions have the property to be inclusion monotonic (i.e.,  $X_{\mathbb{I}} \subseteq Y_{\mathbb{I}} \rightarrow F(X_{\mathbb{I}}) \subseteq F(Y_{\mathbb{I}})$ ), hence allowing the checking of fixpoints.

Unfortunately, due to the over-approximation nature of interval analysis, a quick divergence in the reachability calculation may happen. This is mainly due to the following issues [20]:

- *The dependency problem* which is the lack of interval arithmetic to identify different occurrences of the same variable. For example,  $x-x=0$  holds for each  $x \in [1, 2]$ , but  $X-X$  for  $X = [1, 2]$  yields  $[-1, 1]$ .
- *The wrapping effect* which appears when the results of a computation are overestimated when enclosed into intervals, hence leading to error accumulation at each time step.

To overcome the above mentioned drawbacks of interval computation, Taylor model arithmetics were developed recently by Berz *et. al* [3], [18] as an interval extension to Taylor approximations allowing the non-linear approximation of system reachable states using non-convex enclosure sets.

Formally, a Taylor model  $T_f := p_n(x) + I$  for a given function  $f$  consists of a multivariate polynomial  $p_r(\mathbf{x})$  of order  $r$  in  $d$  variables, and a remainder interval  $I$ , which encloses Lagrange remainder of the Taylor approximation. Hence, the Taylor model arithmetics use interval computation to obtain reliable enclosures not only for the error term but also for every term of the series, allowing the computation of an over-approximation of the solution function at each time point. In addition, symbolic simplifications are applied at each step, hence reducing the interval calculations and consequently delaying divergence problems, usually, associated with interval based techniques.

**Definition 4: Taylor Model**  $T_f := (P_{r,f}, I_{r,f})$  is called a Taylor model of order  $r$  of a function  $f \Leftrightarrow \forall x \in X : f(x) \in P_{r,f}(x-x_0) + I_{r,f}$ , where  $X$  is an interval,  $P_{r,f}(x-x_0)$  is a Taylor approximation polynomial of order  $r$  around the point  $x_0$ . An interval  $I_{r,f}$  is called a remainder bound of order  $r$  of  $f$  on  $X \Leftrightarrow \forall x \in X : R_{r,f}(x-x_0) \in I_{r,f}$ .

The basic arithmetic rules on Taylor models are defined as follows [3], [18]:

- **Addition:**  $T_{r,f+g} := T_{r,f} + T_{r,g} \triangleq (P_{r,f} + P_{r,g}, I_{r,f} + I_{r,g})$
- **Scalar multiplication:**  $T_{r,\alpha f} = \alpha T_{r,f} \triangleq (\alpha P_{r,f}, \alpha I_{r,f})$ , ( $\alpha \in \mathbb{R}$ )
- **Multiplication:**  $T_{r,fg} \triangleq T_{r,f} T_{r,g} := (P_{r,fg}, I_{r,fg})$

with:

- $P_{r,f} P_{r,g} = P_{r,fg} + P_e$
- $P_e \in I_{P_e}$
- $P_{r,f} \in I_{P_{r,f}}$
- $P_{r,g} \in I_{P_{r,g}}$
- $I_{r,fg} := I_{P_e} + I_{P_{r,f}} I_{r,g} + I_{r,f} (I_{P_{r,g}} + I_{r,g})$

Based on the above rules, the Taylor model method extends mathematical operations and functions to Taylor models such that the inclusion relationships are preserved. This is demonstrated by the following theorem:

**Theorem 2:** [18] Let  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  be a continuous function, and  $f \in T$ , where  $T$  is the Taylor model of  $f$ , then  $T \subseteq F$ , where  $F$  is the inclusion function of  $f$ . Moreover, for two functions  $f_1 \in T_1$  and  $f_2 \in T_2$ , we have  $(f_1 + f_2) \in T_S$  and  $(f_1 \cdot f_2) \in T_P$ , where  $T_S$  and  $T_P$  are Taylor models for the sum and product of  $T_1$  and  $T_2$ , respectively.

In practice, the evaluation of a function is transformed to symbolically computing the Taylor polynomial  $p_r(x)$  of the function, which will be propagated throughout the evaluation steps, thus hardly affected by the dependency problem or the wrapping effect. Only the interval remainder term and polynomial terms of orders higher than  $r$ , which are usually small, are bounded using intervals as described by the rules mentioned above and are processed according to the rules of interval arithmetic. This will be demonstrated by the following example:

*Example 3:* In non-linear analog circuits, voltages and currents can be described using analytic functions. For example, in a *BJT* transistor [10], the collector current is described as  $i_C = I_{SE} \frac{V_{BE}}{V_T} (1 + \frac{V_{CE}}{V_A})$ , with  $V_{CE}$  is the output voltage of a differential stage. In such case,  $V_{CE} = \tanh(y) + K$ , where  $K$  is an arbitrary voltage. Consider the Taylor models  $T_1$  and  $T_2$  of the functions  $e^x$ , and  $\tanh(y)$ , respectively, where  $x = \frac{V_{BE}}{V_T}$ , the multiplication  $e^x \tanh(y)$  can be done using Taylor model arithmetic of two Taylor models of order 3. Let  $x, y \in W = [-0.693, 0.693]$  and  $T_1(x) := 1 + x + \frac{x^2}{2} + [-0.11, 0.11]$  and  $T_2(y) := y - \frac{y^3}{3} + [-0.108, 0.108]$ . It holds that:

$$\begin{aligned} T_1(x)T_2(y) &\in (1 + x + \frac{x^2}{2})(y - \frac{y^3}{3}) + (1 + x + \frac{x^2}{2}) \\ &\quad [-0.108, 0.108] + (y - \frac{y^3}{3})[-0.11, 0.11] + \\ &\quad [-0.11, 0.11][-0.108, 0.108] \\ &\subseteq -\frac{1}{6}x^2y^3 - \frac{xy^3}{3} - \frac{y^3}{3} + \frac{x^2y}{2} + xy + y + \\ &\quad (1 + W + \frac{W^2}{2})[-0.108, 0.108] + \\ &\quad (W - \frac{W^3}{3})[-0.11, 0.11] + [-0.22, 0.22] \\ &\simeq -\frac{y^3}{3} + \frac{x^2y}{2} + xy + y + [-0.62, 0.54] \end{aligned}$$

In order to deal with the discrete part of the AMS design, as a generalization of the inclusion function, interval analysis provides efficient and safe methods for checking truth values of Boolean propositions over intervals by using the notion of *inclusion test*.

**Definition 5: Inclusion Test.** Given a constraint  $c : \mathbb{R}^d \rightarrow \mathbb{B}$ , we define  $C_{\mathbb{I}} : \mathbb{I}^d \rightarrow \mathbb{B}_{\mathbb{I}}$  to be an inclusion test of  $c$ , with an interval domain defined with three values set;  $\mathbb{B}_{\mathbb{I}} = \{0, 1, [0, 1]\}$ , where 0 stands for false, 1 for true and  $[0, 1]$  for indeterminate, iff:

$$\{c(x_1, \dots, x_d) | x_1 \in X_1, \dots, x_d \in X_d\} \subseteq C_{\mathbb{I}}(X_1, \dots, X_d)$$

where  $X_i \in \mathbb{I}$ ,  $i \in \{1, \dots, d\}$ .

Inclusion test can be used during the verification algorithm to prove whether the reachable interval states satisfy a given property, or not. We define the inclusion test as follows:  $C_{\mathbb{I}}(X) = 1 \Rightarrow \forall x \in X, c(x) = 1$  and  $C_{\mathbb{I}}(X) = 0 \Rightarrow \forall x \in X, c(x) = 0$ . For instance, given a set of reachable interval states and a property predicate, we remove the states that do not satisfy

$$1 \in C_{\mathbb{I}}(X_1, \dots, X_n)$$

Therefore, if  $C_{\mathbb{I}}(X_1, \dots, X_n) = \emptyset$ , then we have a guarantee that the property is not satisfied.

Let  $x_{\mathbb{I}} = [a, b]$  and  $y_{\mathbb{I}} = [a', b']$  be two real intervals (bounded and closed), a set of the main logical rules that define the inclusion test is given as follows:

$$\begin{aligned} x_{\mathbb{I}} \leq^t y_{\mathbb{I}} = 1 &\Leftrightarrow b \leq a' \\ x_{\mathbb{I}} \in^t y_{\mathbb{I}} = 1 &\Leftrightarrow x_{\mathbb{I}} \in y_{\mathbb{I}} \\ &\Leftrightarrow a \geq a' \text{ and } b \leq b' \end{aligned}$$

*Example 4:* Consider the switching condition in the circuit of Figure 3 defined as  $Cond_1 := v_{c1}(n-1) \leq v_{c2}(n-1)$ , then we have the following:

$$\begin{aligned} (v_{c1}(n-1) := [1, 3]) \leq (v_{c2}(n-1) := [3, 5]) &= 1 \\ (v_{c1}(n-1) := [1, 3]) \leq (v_{c2}(n-1) := [2, 5]) &= [0, 1] \end{aligned}$$

## V. INTERVAL BASED BOUNDED MODEL CHECKING

In this section, we present bounded model checking (BMC) algorithm to support AMS designs. We explore a solution relying on symbolic and interval computational methods. Our BMC approach is based on modeling the transition function as SREs over the Taylor models forms. We proceed on the SREs traces using a time step  $h$  which implies that our answer is relative to a limited time interval. For recurrence equations, we have  $h = 1$ . For differential equations, we approximate them using Taylor model with  $h \in \mathbb{R}$ , ensuring the accumulated error due to  $h$ -approximation is confined in the Interval part of the Taylor model. We consider properties specified in a LTL like language.

In the remaining of this section we will describe the symbolic simulation, and the property checking algorithm of the proposed methodology.

### A. The Symbolic Simulation Algorithm

The generation of the SREs and the evaluation of Taylor model forms rely on rewriting rules based on the symbolic simulation algorithm developed in [1] for digital systems and extended for discrete-time AMS designs in [2]. The symbolic simulation algorithm *ReplaceRepeated(Expr, R)* shown in Algorithm 1 is based on rewriting by repetitive substitution, which applies recursively a set of rewriting of rules  $R$  on an expression  $Expr$  until a fixpoint is reached.

---

#### Algorithm 1 *ReplaceRepeated(Expr, R)*

---

- 1:  $Expr = expr$
  - 2: **repeat**
  - 3:  $Expr_t = ReplaceList(Expr, R)$
  - 4:  $Expr = Expr_t$
  - 5: **until**  $FP(Expr_t, R)$
- 

**ReplaceList(Expr, R):** The substitution function *ReplaceList* takes as arguments an expression  $Expr$  and a list of substitution rules  $R = \{R_1, R_2, \dots, R_n\}$ . It applies each rule sequentially on the expression.

**FP(Expr, R):** A substitution fixpoint  $FP(Expr, R)$  is obtained, if:

$$ReplaceList(expr, R) \equiv ReplaceList(ReplaceList(expr, R), R)$$

The correctness of this rewriting algorithm as well as the proof of termination and confluence of the rewriting system is discussed in [1].

Depending on the type of expressions, we distinguish the following kinds of rewriting rules over Boolean and Real domains:

- *Polynomial symbolic expressions*:  $R_{Math}$  for the simplification of polynomial expressions ( $\mathbb{R}^n[x]$ ).
- *Logical symbolic expressions*:  $R_{Logic}$  for the simplification of Boolean expressions and to eliminate obvious ones like  $(and(a, a) \rightarrow a)$  and  $(not(not(a)) \rightarrow a)$ .
- *If-formula expressions*:  $R_{IF}$  for the simplification of computations over If-formulae. The definition and properties of the  $IF$  function, like reduction and distribution, are used.
  - IF Reduction:  $IF(x, y, y) \rightarrow y$
  - IF Distribution:  $f(A_1, \dots, IF(x, y, z), \dots, A_n) \rightarrow IF(x, f(A_1, \dots, y, \dots, A_n), f(A_1, \dots, z, \dots, A_n))$
For example  $a + IF(x > 0, b, a) \rightarrow IF(x > 0, b + a, a + a)$

For Taylor model generation and evaluation over intervals, we used the following rules which were developed based on the properties described in Section IV

- *Taylor expressions*:  $R_{Tlr}$  are rules intended for the simplification of Taylor model expressions ( $T_{r,f}$ ).
- *Interval expressions*:  $R_{Int}$  are rules intended for the simplification of interval expressions.
- *Interval-Logical symbolic expressions*:  $R_{Int-Logic}$  are rules intended for the simplification of Boolean expressions over intervals.

### B. Temporal Properties

We use an LTL like syntax to represent the properties. The syntax is composed of formulae  $P(n)$  defined recursively and built using Boolean expressions over atomic propositions with temporal operators: eventually  $F$  and always  $G$ . To describe properties on analog signals like current and voltages, atomic propositions encode predicates (inequalities) over reals;  $p(n) \sim c$ , where  $p(n)$  is a polynomial over the state variables,  $\sim \in \{<, \leq, >, \geq, =, \neq\}$ ,  $c \in \mathbb{R}$ . As in traditional BMC, we define temporal operators regarding a bounded time step  $k$ .

**Always operator G**:  $\mathbf{GP}(n)$  specifies that a property  $P(n)$  holds in the current time step of a given path iff the property and the operand hold at the current state and all previous states. Iteratively, we write:

$$\mathbf{GP}(n+1) = \bigwedge_{k=1}^{n+1} P(k)$$

**Eventually operator F**:  $\mathbf{FP}(n)$  specifies that a property holds at the current state or at a previous state. Iteratively, we write:

$$\mathbf{FP}(n+1) = \bigvee_{k=1}^{n+1} P(k)$$

In fact, the inverse of the property ( $\neg P$ ) under verification is used in the BMC algorithm. When a satisfying valuation is

returned by the solver, it is interpreted as a counterexample of length  $k$  and the property  $P$  is proved satisfied ( $\neg P$  is satisfied). However, if the problem is determined to be unsatisfiable, the solver produces a proof (of unsatisfiability) of the fact that there are no counterexamples of length  $k$ .

### C. Verification Algorithm

The bounded forward reachability algorithm starts at the initial states and at each step computes the image, which is the set of reachable interval states. This procedure is continued until either the property is falsified in some state or no new states are encountered. We define the interval based transition system denoting the behavior of the system as follows:

**Definition 6: Interval based state machine.** An *Interval based state machine* is a tuple  $\mathcal{T}_I = (S_I, S_{I,0}, \rightarrow_{T_f})$ , where  $S_I$  is the interval state space,  $S_{I,0} \subseteq S_I$  is the set of initial interval states,  $\rightarrow_{T_f} \subseteq S_I \times S_I$  is a relation defined using Taylor model forms  $T_f$  and capturing the abstract transition between interval states such that:

$$\{s \rightarrow_{T_f} s' \mid \exists a \in s, \exists b \in s' : b = f(a) \text{ and } f \in T_f\}$$

where  $a, b \in \mathbb{R}^d$ ,  $s, s' \in S_I$ ,  $f = \{f_1, \dots, f_d\}$ ,  $T = \{T_{f_1}, \dots, T_{f_d}\}$  with  $f_i : \mathbb{R}^d \rightarrow \mathbb{R}$  is a continuous function,  $i \in \{1, \dots, d\}$  and  $f_i \in T_{f_i}$ , where  $T_{f_i}$  is the Taylor model of  $f_i$ .

Bounded model checking over interval domains is then defined as follows:

**Definition 7: BMC** Given a natural number  $k \geq 0$ , an interval based state machine  $(S_I, S_{I,0}, \rightarrow_{T_f})$  as defined above, and a property  $Prop$ , we say that property  $Prop$  is verified for  $k$  steps if:

$$\forall s \in \mathcal{R}^k(S_0) : s \models Prop$$

where  $S_0$  is the set of initial states.

The different steps for checking safety properties is shown in Algorithm 2. The system equations and the (negated) property  $\neg Prop[n]$  to be verified are given, with the equations initialized are provided. The loop in lines (1-13) describes the verification procedure for  $N_{max}$  time steps. At each step  $n$ , we use an evaluation over the Taylor model forms (line 2) to check whether the property is satisfied or not (line 3). If  $\neg Prop[n]$  is satisfied then a counterexample is generated (line 10), if not, then check for fixpoint (line 5), otherwise update the reachable states (line 12) and go to the next time step verification.

**SRE(A, r)**: Given an AMS system  $(A)$  and an order  $r$ ,  $SRE(A, r)$  returns the generalized SREs by applying the symbolic rules described earlier. For the case of a continuous function, the Taylor approximation of order  $r$  is applied to

---

**Algorithm 2** Safety Verification
 

---

**Require:**  $x[n] := SRE(\mathcal{A}, r)$

**Require:**  $\neg Prop[n]$

**Require:**  $\mathcal{R}^0 = S_0$

```

1: for  $n = 1$  to  $N_{max}$  do
2:    $T_{o_t, x[n]} := TM\_Form(x[n], o_t)$ 
3:   if  $Prop\_Check(\neg Prop[n], T_{o_t, x[n]}) == False$  then
4:     if  $Reach[T_{o_t, x[n]}] \subseteq \mathcal{R}^{n-1}$  then
5:       return fixpoint reached
6:     else
7:       Inc_Step(n)
8:     end if
9:   else
10:    Generate_CE
11:  end if
12:   $\mathcal{R}^{n-1} = Update\_Reach(\mathcal{R}^{n-2}, Reach[T_{o_t, x[n-1]}])$ 
13: end for

```

---

generate the SREs.

**TM\_Form**( $\mathbf{x}[n], o_t$ ): Given a set of SREs, *TM\_Form* returns the corresponding Taylor model with order  $o_t$  at the specified time step. Such model will be checked against properties for satisfaction using *Prop\_Check*

**Prop\_Check:** Given the Taylor model forms representing the transition function and the property  $\neg Prop()$ , apply algebraic decision procedures to check for satisfiability. The safety verification at a given step  $n$  can be defined with the following formula:

$$Prop\_Check \triangleq \mathbf{x}[n] = T_{o_t, x[n]}(\mathbf{x}[n-1], h) \wedge \neg Prop(\mathbf{x}[n])$$

**Reach**[ $T_{o_t, x[n]}$ ]: Given the Taylor model form at an arbitrary time step, **Reach** evaluates the reachable states at according to the following definition:

*Definition 8: 1-Step Reachable states.* The set of reachable states in 1 step from a given set of states  $S_k \subseteq \mathbb{I}^d$ , is denoted by  $\mathcal{R}_1(S_k)$  and is defined as:

$$\mathcal{R}_1(S_k) \triangleq \{s' \in S_{k+1} | \exists s \in S_k : \vec{F}_1(s) = s'\}$$

where  $S_{k+1} \subseteq \mathbb{I}^d$ ,  $\vec{F} = (F_1, \dots, F_d)$ , with  $F_i : \mathbb{I}^d \rightarrow \mathbb{I}$  is an interval evaluation of Taylor model form of the function  $f_i : \mathbb{R}^d \rightarrow \mathbb{R}$ ,  $i \in \{1, \dots, d\}$ .

**Update\_Reach**( $R_1, R_2$ ): The function returns the union of the states in the sets  $R_1$  and  $R_2$  according to the following definition:

*Definition 9:* The set of reachable states in less than  $k$  steps ( $0 < l < k$ ), from a given set of  $S_0$  of states, is denoted by  $\mathcal{R}^{<k}(S_0)$ , and is defined as:

$$\mathcal{R}^{<k}(S_0) \triangleq \bigcup_{l < k} \mathcal{R}^l(S_{l-1})$$

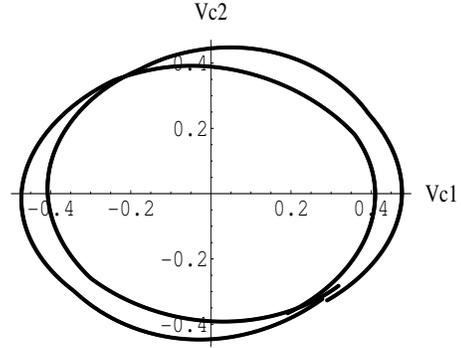


Fig. 4. Oscillation Behavior for Circuit in Example 2

For instance, in Algorithm 2 (line 12) we have  $\mathcal{R}^{n-2} = \mathcal{R}^{<n-1}(S_0)$  and  $Reach[T_{o_t, x[n-1]}] = R_1(s_{n-1})$

## VI. EXPERIMENTAL RESULTS

We have applied the verification algorithm on the analog circuit described in Example 2 and a third-order  $\Delta\Sigma$  modulator. For the analog circuit, we checked the oscillation property for given set of initial voltages and for all switching conditions. We formally describe the oscillation property as:

$$Prop_1 : \mathbf{G}((p_1 \Rightarrow \mathbf{F}p_2) \wedge (p_2 \Rightarrow \mathbf{F}p_1))$$

where  $p_1 = \neg p_2 := V_{c1} < V_{c2}$ . For instance, when the analog circuit is described by  $v_{c1} = v_{c2}$  and  $v_{c2} = -v_{c1} + v_{c1}^3$ , the reachable states for the oscillation behavior are shown in Figure 4, bounded by the corresponding Taylor model. We also checked several safety properties, e.g.,

$$Prop_2 : \mathbf{G}(-0.5 < V_{c1} < 0.5) \wedge (-0.5 < V_{c2} < 0.5)$$

and

$$Prop_3 : \mathbf{G}(-1 < V_{c2} < 1)$$

Details on initial conditions are shown below:

$$Parameters_{s_1} \rightarrow \begin{cases} a \rightarrow [-0.03, 0.03] & b \rightarrow [-0.03, 0.03] \\ h \rightarrow 0.01 \\ x[0] = 0.3 + a & y[0] = -0.3 + b \end{cases}$$

$$Parameters_{s_2} \rightarrow \begin{cases} a \rightarrow [-0.03, 0.03] & b \rightarrow [-0.03, 0.03] \\ h \rightarrow 0.01 \\ x[0] = 1 + a & y[0] = 0.2 + b \end{cases}$$

The verification results for 2 possible switching cases of this circuit (we refer to as circuit 1 and circuit 2) are shown in Table I. For the first set of initial conditions, we find that the circuit is behaving in accordance with the properties, hence the properties are satisfied. For the second set of initial conditions, the safety properties  $Prop_2$  and  $Prop_3$  are violated while divergence prevent us to check whether the

circuits are oscillating or not <sup>1</sup>.

Table II shows the verification results for a third-order  $\Delta\Sigma$  modulator <sup>2</sup>. A  $\Delta\Sigma$  modulator is said to be stable if the integrator output remains bounded under a bounded input signal, thus avoiding that the quantizer in the modulator becomes overloaded which leads to instability. The stability properties is written as:  $P(k) := \mathbf{G}(-1 < x3(k) < 1)$ , where  $x3$  is the input to the quantizer. For a first set of initial constraints, the modulator loses stability after 5 time steps. For the second, set of constraints the modulator was proved stable for 45 time steps.

TABLE II  
VERIFICATION RESULTS FOR 3<sup>rd</sup> ORDER  $\Delta\Sigma$  MODULATOR

Initial Constraints	Property Evaluation for $n = 0$ to $N_{max}$ Cycles	CPU time Used
Initial constraints 1	$N_{max} = 15$ $n = 0$ to $5$ True $n > 5$ False	3.89 sec
Initial constraints 2	$N_{max} = 45$  True	120.8 sec

## VII. RELATED WORK

We can identify two classes of verification techniques for AMS designs, namely state exploration methods (e.g., reachability analysis) and algebraic methods (e.g., constraint solving). Common to the proposed state based methods is the necessity of the explicit computation of either exact or approximate reachable sets corresponding to the continuous dynamics, hence deducing properties about the properties of the design under verification. In the constraint based methods, the AMS design is described by a set of equations (algebraic or difference equations) along with a set of constraints. Algebraic and logical rules are then applied to check whether the system satisfies such constraints or not.

For instance, model checking and reachability analysis were proposed for validating AMS designs over a range of parameter values and a set of possible input signals. Several methods for approximating reachable sets for continuous dynamics have been proposed in the open literature. They rely on the discretization of the continuous state space by using over-approximating domains like polyhedra and intervals. In [15], the authors construct a finite-state discrete abstraction of electronic circuits by partitioning the continuous state space into fixed size hypercubes and computed the reachability relations between these cubes using numerical techniques. In [11], the authors tried to overcome the expensive computational method in [15], by combining discretization and projection techniques of the state space, hence reducing its dimension. While the approach in [11] is less precise due to the use of projection techniques, it is still sound. Variant approaches of the latter analysis were proposed. For instance, the model checking

tools *d/dt* [6], *Checkmate* [9] and *PHaver* [8] were adapted and used in the verification of a biquad low-pass filter [6], a tunnel diode oscillator and a  $\Delta\Sigma$  modulator [9], and voltage controlled oscillators [8]. In [13], the authors used intervals to construct the abstract state space, while used heuristics to identify possible transition between adjacent regions, main difference with [15], is that they allow variable sized regions. Petri nets based models and algorithms have been developed also for the reachability analysis of AMS designs in [17], [16].

The AMS verification we present in this paper is in the same spirit as the above mentioned works in terms of requirement for state exploration. However, we can identify two distinct points. First, we rely on a recurrence equation form as a way to model the design rather than automata, which provide us with more compact representation. Second, we apply the verification over Taylor model forms which provide tight bounds for the reachable states by using non-convex over approximation. In addition, Taylor models allow the symbolic representation of the reachable states using polynomials terms, therefore minimizing the risk of state explosion.

In [12], the author proposed an approach for specifying and reasoning about digital systems that are described at the analog level of abstraction. The approach relies upon specifying the behaviors of analog components by piecewise-linear predicates on voltages and currents. Theorem proving and constraint based methods are then used to check for the implication relation between the implementation and the specification. In [7], the authors developed a bounded model checking prototype tool (*Property-Checker*) for the verification of the static behavior of AMS designs. The basic idea is based on validity checking of first-order formulae over a finite interval of time. In [7], the authors trade-off accuracy with efficiency by basing the analysis on rational numbers rather than real numbers, however affecting the soundness of the verification. In addition to the loose approximations in [12], [7], the verification is only possible for static behavior.

In [2], [23], the authors propose an induction verification approach for AMS designs using symbolic methods. The procedure is iterative in the sense that if the proof is obtained, then the property is verified. Otherwise, generated counterexamples are analyzed and constraints refinement is applied and verification is repeated until the property is verified or a concrete counterexample is identified. Such methodology is limited for AMS systems that can be described using discrete time models, while our approach consider continuous time systems. More details about the application of formal methods to the verification of AMS designs can be found in [24].

## VIII. CONCLUSION

In this paper, we have defined a bounded model checking approach for AMS systems modeled in terms of combination between SRE and differential equations. We have proposed a semi-symbolic modeling of the state space using the principle of Taylor models which provide a way for representing a combination of representation using a combination of polynomials and interval terms. The main advantage of such modeling is

<sup>1</sup>The experiments were performed on Intel Core2 1900 MHz processor and 2GB of RAM

<sup>2</sup>Details about the design can be found in [22]

TABLE I  
VERIFICATION RESULTS

Circuit & Properties	BMC Verification for $k = 0$ to $N_{max}$ Steps	CPU & Memory Used
Circuit 1 (Parameters 1) Oscillation Property <i>Prop<sub>2</sub></i> <i>Prop<sub>3</sub></i>	$N_{max} = 700$ <i>Proved True</i> <i>Proved True</i> <i>Proved True</i>	107.39 sec 7.93 MB
Circuit 1 (Parameters 2) Oscillation Property <i>Prop<sub>2</sub></i> <i>Prop<sub>3</sub></i>	$N_{max} = 700$ Not Verified (Divergence) <i>Proved False</i> at $k = 18$ <i>Proved False</i> at $k = 18$	108.41 sec 7.14 MB
Circuit 2 (Parameters 1) Oscillation Property <i>Prop<sub>2</sub></i> <i>Prop<sub>3</sub></i>	$N_{max} = 1200$ <i>Proved True</i> <i>Proved True</i> <i>Proved True</i>	583.75 sec 51.15 MB
Circuit 2 (Parameters 2) Oscillation Property <i>Prop<sub>2</sub></i> <i>Prop<sub>3</sub></i>	$N_{max} = 1200$ Not Verified (Divergence) <i>Proved False</i> at $k = 4$ <i>Proved False</i> at $k = 9$	584.05 sec 50.60 MB

the fact, that the polynomial representation helps slowing the divergence due to the over-approximated intervals, meanwhile, the interval part provides an important abstraction to handle the continuous behavior.

We have developed and implemented this arithmetic as a set of simplification rules a the bounded model checking algorithm. Experimental results have proven the feasibility and the utility of the approach. However, the method is still limited in terms of capacity. In fact, we have implemented our methodology using standard libraries for symbolic computation available in *Mathematica*.

Future research directions include investigating alternative implementations to improve the experimental capacity over more complex systems and to measure the limitation of the proposed methodology. Also, an important effort is needed to classify the kind of properties and AMS systems that can be verified using this verification approach.

## REFERENCES

- [1] G. Al-Sammame. Simulation Symbolique des Circuits Decrits au Niveau Algorithmique. PhD thesis, Université Joseph Fourier, Grenoble, France, July 2005.
- [2] G. Al Sammane, M. Zaki, and S. Tahar. A Symbolic Methodology for the Verification of Analog and Mixed Signal Designs. *IEEE/ACM Design Automation and Test in Europe*, pp. 249-254, 2007.
- [3] M. Berz, G. Hoffsttter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds, *Reliable Computing*, 4(1): 83-97, Springer, 1998.
- [4] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
- [5] E. M. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Computational challenges in bounded model checking. *International Journal on Software Tools for Technology Transfer*, 7(2): 174-183, Springer, 2005.
- [6] T. Dang, A. Donze, O. Maler. Verification of Analog and Mixed-signal Circuits using Hybrid System Techniques. In *Formal Methods in Computer-Aided Design*, LNCS 3312, pp. 14-17, Springer, 2004.
- [7] M. Freibothe, J. Schoenherr, and B. Straube. Formal Verification of the Quasi-Static Behavior of Mixed-Signal Circuits by Property Checking. *Electr. Notes Theor. Comput. Sci.*, 153(3):23-35, Elsevier, 2006.
- [8] G. Frehse, B.H. Krogh, R.A. Rutenbar. Verifying Analog Oscillator Circuits Using Forward/Backward Abstraction Refinement. *IEEE/ACM Design Automation and Test in Europe*, pp. 257-262, 2006.
- [9] S. Gupta, B.H. Krogh, R.A. Rutenbar. Towards Formal Verification of Analog Designs, *IEEE/ACM International Conference on Computer Aided Design*, pp. 210-217, 2004.
- [10] P.R. Gray, P.J. Hurst, S.H. Lewis, and R.G. Meyer. *Analysis and Design of Analog Integrated Circuits*, Wiley, 2001
- [11] M. R. Greenstreet, I. Mitchell. Reachability Analysis Using Polygonal Projections. In *Hybrid Systems: Computation and Control*, LNCS 1569, pp. 103-116, Springer, 1999.
- [12] K. Hanna. Automatic Verification of Mixed-Level Logic Circuits. In *Formal Methods in Computer-Aided Design*, LNCS 1522, pp.133-166, Springer, 1998.
- [13] W. Hartong, R. Klausen, and L. Hedrich. Formal Verification for Non-linear Analog Systems: Approaches to Model and Equivalence Checking, *Advanced Formal Verification*, pp. 205-245, Kluwer, 2004.
- [14] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1(1-2):110-122, Kluwer, 1997.
- [15] R.P. Kurshan and K.L. McMillan. Analysis of Digital Circuits Through Symbolic Reduction. *IEEE Trans. on Computer-Aided Design*, 10(11): 1356-71, 1991.
- [16] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda. Verification of Analog/mixed-signal Circuits using Labeled Hybrid Petri Nets, *IEEE/ACM International Conference on Computer Aided Design*, pp. 275-282, 2006.
- [17] S. Little, D. Walter, N. Seegmiller, C. Myers, and T. Yoneda. Verification of Analog and Mixed-Signal Circuits Using Timed Hybrid Petri Nets. In *Automated Technology for Verification and Analysis*, LNCS 3299, pp. 426-440, Springer, 2004.
- [18] K. Makino, M. Berz. Remainder Differential Algebras and their Applications. In *Computational Differentiation: Techniques, Applications, and Tools*, pp. 63-75, SIAM, 1996.
- [19] C. J. Myers, R. R. Harrison, D. Walter, N. Seegmiller, S. Little. The Case for Analog Circuit Verification. *Electr. Notes Theor. Comput. Sci.*, Elsevier, 153(3):53-63, 2006.
- [20] R.E. Moore. *Methods and Applications of Interval Analysis*, Society for Industrial and Applied Mathematics, 1979.
- [21] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley Longman Publishing, USA, 1991.
- [22] M. Zaki, G. Al Sammane, S. Tahar, and G. Bois. A Bounded Model Checking Approach for AMS Designs. Technical Report, ECE Dept., Concordia University, Montreal, Quebec, Canada, May 2007. [http://hvg.ece.concordia.ca/Publications/TECH\\_REP/AMS\\_BMC\\_TR07](http://hvg.ece.concordia.ca/Publications/TECH_REP/AMS_BMC_TR07)
- [23] M. Zaki, G. Al Sammane and S. Tahar. Formal Verification of Analog and Mixed Signal Designs in Mathematica, *Proc. International Conference on Computational Science*, LNCS 4488, pp. 263-267, Springer, 2007.
- [24] M. Zaki, S. Tahar, and G. Bois. Formal Verification of Analog and Mixed Signal Designs: Survey and Comparison, *IEEE Northeast Workshop on Circuits and Systems*, pp. 281-284, 2006.

# Analyzing Gene Relationships for Down Syndrome with Labeled Transition Graphs

Neha Rungta\*, Hyrum Carroll\*, Eric G Mercer\*, Randall J. Roper†, Mark Clement\*, Quinn Snell\*

\*Computer Science Department, Brigham Young University, Provo, UT, USA

Email: {neha, hdc, egm, clement, snell}@cs.byu.edu

†Department of Biology and Indiana University Center for Regenerative Biology and Medicine,  
Indiana University-Purdue University Indianapolis, Indianapolis, IN, USA

Email: rjroper@iupui.edu

**Abstract**—The relationship between changes in gene expression and physical characteristics associated with Down syndrome is not well understood. Chromosome 21 genes interact with non-chromosome 21 genes to produce Down syndrome characteristics. This indirect influence, however, is difficult to empirically define due to the number, size, and complexity of the involved gene regulatory networks. This work links chromosome 21 genes to non-chromosome 21 genes known to interact in a Down syndrome phenotype through a reachability analysis of labeled transition graphs extracted from published gene regulatory network databases. The analysis provides new relations in a recently discovered link between a specific gene and Down syndrome phenotype. This type of formal analysis helps scientists direct empirical studies to unravel chromosome 21 gene interactions with the hope for therapeutic intervention.

## I. INTRODUCTION

Researchers currently hypothesize that cancers and other physical ailments are caused by duplication of genes on chromosomes [1]. Chromosomes are made up of genes that encode information about physical characteristics or phenotypes. Through empirical analysis, systems biologists determine how genes are linked to certain phenotypes. Knowing how genes interact to express specific phenotypes is a critical step for therapeutic intervention in many diseases. The large number of genes found in an individual and the possible gene interactions make it difficult to manually discern how gene regulatory networks influence phenotypic characteristics.

Patients with Down syndrome (DS) have an extra copy of chromosome 21 (chr. 21) and have phenotypes like abnormal brain and facial features as well as mental retardation. DS results from three copies of approximately 350 chr. 21 genes and has many well defined phenotypes that make it an excellent model to understand gene regulatory networks. By understanding changes that occur in this model, gene regulatory networks in other diseases that are not as well-defined genetically or phenotypically can be better understood.

Phenotypes are generated by interactions between genes that are described by gene regulatory networks. Each gene regulatory network is a complex feedback circuit that is constructed through large, costly, and time consuming empirical studies. The regulatory networks once understood, however, provide a wealth of information. For example, pharmaceutical researchers use gene regulatory network diagrams to design

compounds that inhibit the expression of certain genes. In other words, pharmaceutical researchers are designing drugs to target genes involved in specific medical conditions; the drugs essentially manipulate the gene regulatory networks to produce a desired outcome.

DS researchers and systems biologists believe that changes in expression levels of one or more genes on the extra copy of chr. 21 lead to specific DS phenotypes. Even though DS researchers and bioinformaticians have documented elevated expression levels of chr. 21 genes in DS, they have been unable, for the most part, to directly link a specific chr. 21 gene to a particular phenotype.

Research in human and mouse models shows that there are modifier genes, not found on chr. 21, that contribute to DS phenotypes [2], [3]. For example, the *Sonic hedgehog* gene (SHH) in the Hedgehog signaling pathway has been directly linked to a DS brain phenotype [2]. It is, however, unknown how the SHH gene is linked to chr. 21 genes. The work in this paper helps researchers focus on specific modifier and chr. 21 genes for empirical studies.

Researchers often relate chr. 21 genes to modifier genes by manually examining published gene regulatory network databases. These databases, [4]–[6], are exceptionally complex and regularly evolve as new data is discovered through empirical studies. Manual extraction of indirect connections between chr. 21 and modifier genes is not feasible in such a large, concurrent, and connected system. An automated formal analysis is required to assist researchers in understanding gene regulatory networks important in DS.

We use a formal approach to analyze gene regulatory networks that are mined from different biological databases. We build a single labeled transition graph from these databases by defining a way of connecting the gene regulatory networks. We perform an exhaustive reachability analysis on the graph using a randomized breadth-first search (BFS). Randomized BFS gives a sample of shortest path connections between modifier genes and chr. 21 genes. The random paths are tabulated and graphed to show researchers the names and frequencies of genes found in these paths. We hypothesize that genes frequently found in paths connecting modifier and chr. 21 genes are more likely to be involved in a specific phenotype. Our results demonstrate a possible relationship between a non-

chr. 21 modifier gene and chr. 21 genes. These relationships help DS researchers to direct resources for future empirical studies.

The principle contributions of this work are: (1) a biologically feasible technique connecting different gene regulatory networks into a single labeled transition graph suitable for formal analysis; (2) a reachability analysis using randomized BFS to generate different traces between chr. 21 genes and modifier genes; and (3) tabulated results showing potential interactions between specific chr. 21 genes and the SHH modifier gene.

## II. RELATED WORK

Several databases store pictorial representations of empirically curated gene regulatory networks [4]–[6]. Among all the databases, KEGG [4], currently has the largest amount of data in the most comprehensible and accessible format. It provides 175 pathways with over 12,000 genes from the human genome. Many of its pathways include both a pictorial and XML representation; although, the XML descriptors often have fewer defined interactions than those defined in the pictorial descriptors. Another source of gene interactions is found in a PubMed abstracts database [7]. PubMed is a premier journal index for published bio-medical articles. Existing research, [7], extracts gene relationships from PubMed using natural language processing algorithms. The work in this paper uses the KEGG database and the PubMed extraction to build the labeled transition graph of the gene regulatory networks.

State of the art pathway analyses tools such as Cytoscape [8], Reactome [9], and Bind [10] visualize gene regulatory networks. They provide some basic query mechanisms to probe structure in the regulatory networks. The queries, however, are simple and do not provide an ability to derive indirect relationships between modifier and chr. 21 genes in a fully automated manner.

Other formal verification approaches that analyze regulatory networks consider only a single network or pathway [11]–[14]. A single network is modeled in isolation to predict pathway behavior based on the gene interaction rates. The work in this paper abstracts many details of individual regulatory networks to consider the interactions of the complete system of regulatory networks. Where existing research, [11]–[14], tries to understand intra-network interactions, we extend the analysis to inter-network interactions in expressing a specific phenotype.

## III. GRAPH CONSTRUCTION

We build a labeled transition graph from the KEGG and PubMed databases. The process follows three steps: first, we abstract the reactions and compounds in gene regulatory networks in the KEGG database to create intra-pathway gene interactions; second, we use gene interactions to create inter-pathway connections; and third, we add PubMed interactions to the inter-pathway and intra-pathway connections. Adding the gene-to-gene connections between individual pathways essentially flattens the KEGG database in step two. The final

labeled transition graph is an over approximation of the actual biological system. The process is illustrated in more detail with a simple example.

Fig. 1(a) shows parts of gene regulatory networks and metabolic pathways: the Hedgehog signaling pathway (HSP), Basal cell carcinoma (BCC), and Alzheimer’s disease (AD) regulatory networks. The rectangle boxes in each pathway represent the genes in the regulatory networks. For example, the Hedgehog signaling pathway shown in Fig. 1(a) contains the genes SHH, PTCH, GLI and WNT1. An edge between any two genes is a direct or indirect interaction between the genes. We maintain scalability by abstracting away most of the details in the actual biological system to focus purely on gene interactions—direct and indirect. Results suggest that the abstraction retains enough information to be meaningful. Fig. 1(b) is the final labeled transition graph for the pathways and PubMed relations shown in Fig. 1(a).

The abstract pathways form a set of intra-network gene interactions. Each graph is a separate abstracted regulatory network. For example, the abstract graph of the Hedgehog signaling pathway contributes  $s_0$ ,  $s_1$ ,  $s_4$ , and  $s_5$  nodes to the transition graph in Fig. 1(b). The intra-network connections between the genes in the Hedgehog signaling pathway are maintained in the transformation. The next step connects the abstract networks to form an inter-network graph. The separate abstract networks are connected by creating a set of nodes labeled by both the gene and the network owning the gene. Each node contains the gene label and the owning pathway label. Once the set of nodes is known, then edges are created between nodes that contain common gene labels. For example, Fig. 1(a) shows that both HSP and BCC contain the PTCH gene. As such, Fig. 1(b) connects state  $s_1$  to state  $s_2$  showing a relation between the HSP and BCC pathways through the common PTCH gene. The two different nodes labeled with PTCH gene for the HSP and BCC networks enable us to easily detect that PTCH is the gene which connects the HSP and BCC networks. Connecting regulatory networks in this way essentially flattens the KEGG database through gene interactions.

The final step in building the inter-network graph augments the XML data in the KEGG database with known interactions published in the PubMed database using the work in [7]. In essence, any gene pair related in PubMed is also related in the inter-network graph. For example, the bottom right member of Fig. 1(a) shows a relation defined in the PubMed database; it connects the gene A2M to WNT1. The relation is expressed in Fig. 1(b) by the edge between states  $s_5$  and  $s_6$ .

## IV. ANALYSIS

DS researchers are interested in finding the gene interactions between modifier genes (e.g., SHH gene) and chr. 21 genes. We define a randomized BFS to find shortest path traces between modifier genes and chr. 21 genes.

A regular BFS enumerates all nodes reachable in one-step from the initial node before enumerating nodes reachable in two-steps. This feature guarantees that the BFS finds the

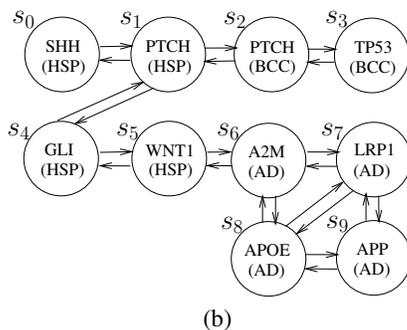
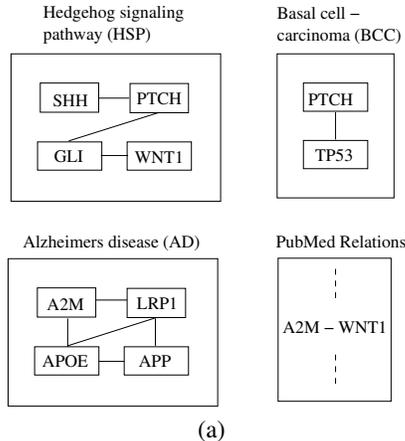


Fig. 1. Converting a set of gene regulatory pathways into a labeled transition graph. (a) KEGG pathways and PubMed Relations. (b) The corresponding labeled transition graph.

shortest distance between the start node and another node of interest. The search, however, deterministically generates the same shortest trace between the two nodes in every single trial. For example, the shortest distance between nodes  $s_6$  and  $s_9$  is two steps in Fig. 1(b). The successors of node  $s_6$  are sorted in some default order. If the successors are lexicographically sorted where node  $s_7$  is inserted in the queue before  $s_8$ , the BFS generates the trace  $s_6 \rightarrow s_7 \rightarrow s_9$  and does not generate the trace  $s_6 \rightarrow s_8 \rightarrow s_9$ .

The labeled transition graph derived from the gene regulatory networks database is a highly connected graph. It has approximately 12,000 nodes and one million edges. About 90% of the connections are a result of the abstraction applied to the pathways while creating the inter-network connections. The high degree of connectivity creates a large number of unique shortest paths between two nodes in the graph. There are different algorithms to find the  $k$  shortest paths between two nodes in a graph such as the one presented by Eppstein in [15]; however, to randomly sample different shortest traces we use a randomized BFS for its algorithmic simplicity and low complexity. Generating  $s_6 \rightarrow s_7 \rightarrow s_9$  is equally likely as generating  $s_6 \rightarrow s_8 \rightarrow s_9$  in any trial of randomized BFS.

A randomized BFS generates a subset of the shortest paths between modifier and chr. 21 genes. The pseudo-code for a

```

procedure Random_BFS( $s_0$ )
1:  $Visited := \{s_0\}$ 
2:  $Queue.enqueue(s_0)$ 
3: while ( $Queue \neq \emptyset$ ) do
4:    $s := Queue.dequeue$ 
5:   if  $is\_chr\_21\_gene(s)$  then
6:     print "Report Path"
7:    $X_{succ} := get\_successors(s)$ 
8:    $s := randomize\_elements(X_{succ})$ 
9:   for each  $s' \in X_{succ}$  do
10:    if  $s' \notin Visited$  then
11:       $Visited := Visited \cup \{s'\}$ 
12:       $Queue.enqueue(s')$ 

```

Fig. 2. Pseudo-code for randomized breadth-first search.

randomized BFS, [16], is presented in Fig. 2. The algorithm is a variant of the regular BFS that randomizes the order of successors before inserting them into a queue. Randomized BFS generates shortest traces of all the gene interactions from the initial state to nodes that represent chr. 21 genes ( $is\_chr\_21\_gene$ ). During the search, when we encounter a node,  $s$ , labeled with a chr. 21 gene (line 5) we report the path from the initial state to the node,  $s$  (line 6). We then continue searching for paths connecting the initial state to nodes labeled with other chr. 21 genes. Note that we maintain a set of visited nodes and never visit the same node twice (lines 10 – 12). If there is more than one shortest path from the initial state to a node with a particular chr. 21 gene, each trial of randomized BFS generates a subset of those traces.

## V. RESULTS

The analysis in this paper is designed to answer the following research questions: (a) How many chr. 21 genes are connected to a modifier gene? (b) What is the length of the shortest path between a chr. 21 gene and the modifier gene? and (c) What are the gene interactions between a chr. 21 gene and the modifier gene? We summarize the analysis for the SHH modifier gene and chr. 21 genes in the following paragraphs.

To answer the research questions (a) and (b), we run a single trial of randomized BFS starting from the SHH modifier gene in the Hedgehog signaling pathway. During the search, whenever we encounter a chr. 21 gene, we mark the chr. 21 gene as reachable from the SHH modifier gene and note the length of the trace. Recall that a randomized BFS returns the length of the shortest trace from the SHH gene to the particular chr. 21 gene.

Randomized BFS finds 38 chr. 21 genes that are between 3 to 7 steps away from the SHH gene in the labeled transition graph. Among the reachable chr. 21 genes, there are genes that play a significant role in Alzheimer's disease and other cancers. This result is especially interesting to researchers because virtually all individuals with DS have indicators of Alzheimer's disease by 40 years of age.

We run several trials of randomized BFS to answer research question (c). The trials result in a large number of traces

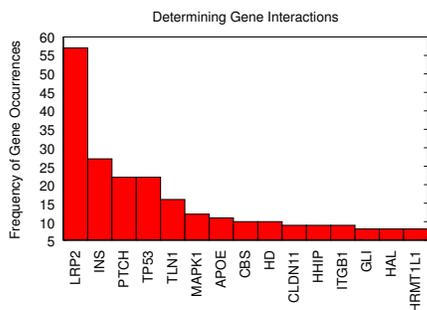


Fig. 3. Gene occurrences in SSH modifier and chr. 21 genes traces.

between the SHH gene and chr. 21 genes. The traces aid us in analyzing the different gene interactions that lead from the SHH gene to chr. 21 genes. For example, consider two arbitrary traces of gene interactions:  $a \rightarrow b \rightarrow c_1$  and  $a \rightarrow b \rightarrow e \rightarrow c_2$ . Since gene  $b$  occurs in both traces it is more likely to affect a certain phenotype. In essence, to answer research question (c), we count the number of times a gene uniquely occurs in multiple traces.

In Fig. 3, for an arbitrary randomized BFS trial, we count the total occurrences of genes found in traces between the SHH modifier gene and chr. 21 genes. The results in Fig. 3 are exciting to biologists due to the high occurrence frequency of LRP2, INS, and MAPK1 genes. These genes have not been previously considered by biologists in the expression of DS phenotypes. The results in Fig. 3 open new avenues for empirical research by providing a list of genes closely related to SHH modifier and chr. 21 genes.

Interestingly, TP53 (involved in cancer) and APOE genes have been previously linked to DS phenotypes by researchers. The fact that our analysis relates TP53 and APOE as frequently occurring in traces between SHH modifier and chr. 21 genes provides anecdotal validation to our results. It gives hope that the other genes, such as LRP2, discovered in this analysis might affect DS phenotypes. Furthermore, genes like PTCH and GLI are part of the Hedgehog signaling pathway and are expected to have a high occurrence in the traces as seen in Fig. 3. This provides further anecdotal evidence that our analysis is biologically sound.

## VI. THREATS TO VALIDITY

The gene regulatory networks in the databases are not currently tagged with phenotype or developmental stage information. This causes the labeled transition graph built from the gene regulatory networks to be an over-approximation of the system. The analysis, however, is aimed to provide biologists a set of genes possibly involved in DS phenotypes. A small set of genes makes it feasible to empirically determine whether the gene affects a DS phenotype. Empirical studies filter any false positives generated during the analysis.

## VII. CONCLUSION AND FUTURE WORK

This work defines a technique to build a labeled transition graph from different biological databases. On this graph we perform a reachability analysis using randomized BFS to find gene interactions between modifier genes and chr. 21 genes. The analysis for the SHH modifier gene and chr. 21 genes gives an interesting set of genes for designing empirical studies. The same analysis can be used for determining gene interactions in various modifier genes for DS and other ailments. The representation of the different gene regulatory pathways as single labeled transition graphs lends itself to a more refined analysis. As future work, biologically interesting questions can be posed in temporal logic to find traces that satisfy the property.

## REFERENCES

- [1] R. Redon *et al.*, "Global variation in copy number in the human genome," *Nature*, vol. 444, no. 7118, 2006.
- [2] R. J. Roper, L. L. Baxter, N. G. Saran, D. K. Klinedinst, P. A. Beachy, and R. H. Reeves, "Defective cerebellar response to mitogenic Hedgehog signaling in Down's syndrome mice," *Proc. Natl Acad Sci*, vol. 103, no. 5, 2006.
- [3] J. R. Arron *et al.*, "NFAT dysregulation by increased dosage of DSCR1 and DYRK1A on chromosome 21," *Nature*, vol. 441, no. 7093, pp. 595–600, 2006.
- [4] "KEGG database." [Online]. Available: <http://www.genome.jp/kegg>
- [5] "Biocyc." [Online]. Available: <http://biocyc.org>
- [6] "Metacyc encyclopedia of metabolic pathways." [Online]. Available: <http://metacyc.org>
- [7] R. Bunescu, R. Ge, R. Kate, E. Marcotte, R. Mooney, A. Ramani, and Y. Wong, "Learning to extract proteins and their interactions from Medline abstracts," in *ICML-2003 Workshop on Machine Learning in Bioinformatics*, August 2003, pp. 46–53.
- [8] P. Shannon *et al.*, "Cytoscape: A software environment for integrated models of biomolecular interaction networks," *Genome Res.*, vol. 13, no. 11, pp. 2498–2504, 2003.
- [9] "Reactome - a curated knowledgebase of biological pathways." [Online]. Available: <http://www.reactome.org/cgi-bin/frontpage>
- [10] G. Bader, I. Donaldson, C. Wolting, B. Ouellette, T. Pawson, and C. Hogue, "Bind—the biomolecular interaction network database," *Nucleic Acids Research*, vol. 29, no. 1, pp. 242–245, 2001.
- [11] J. Heath, M. Kwiatkowska, G. Norman, D. Parker, and O. Tymchyshyn, "Probabilistic model checking of complex biological pathways," in *Proc. Computational Methods in Systems Biology (CMSB'06)*, ser. Lecture Notes in Bioinformatics, C. Priami, Ed., vol. 4210. Springer Verlag, 2006, pp. 32–47.
- [12] H. Kuwahara, C. J. Myers, M. S. Samoilov, N. A. Barker, and A. P. Arkin, "Automated abstraction methodology for genetic regulatory networks," *Transactions on Computational Systems Biology*, vol. 4220, pp. 150–175, 2006.
- [13] M. Kwiatkowska, G. Norman, D. Parker, O. Tymchyshyn, J. Heath, and E. Gaffney, "Simulation and verification for computational modelling of signalling pathways," in *WSC '06: Proceedings of the 38th conference on Winter simulation*. Winter Simulation Conference, 2006, pp. 1666–1674.
- [14] D. L. Dill, M. A. Knapp, P. Gage, C. Talcott, P. Lincoln, and K. Laderoute, "The pathalyzer: a tool for visualization and analysis of signal transduction pathways," in *First Annual RECOMB Satellite Workshop on Systems Biology*, ser. Lecture Notes in Bioinformatics. Springer, 2005.
- [15] D. Eppstein, "Finding the k shortest paths," in *IEEE Symposium on Foundations of Computer Science*, 1994, pp. 154–165. [Online]. Available: [citeseer.ist.psu.edu/eppstein97finding.html](http://citeseer.ist.psu.edu/eppstein97finding.html)
- [16] M. B. Dwyer, S. Person, and S. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM Press, 2006, pp. 92–104.

# A Formal Model of Clock Domain Crossing and Automated Verification of Time-Triggered Hardware

Julien Schmaltz<sup>1</sup>

Institute for Computing and Information Sciences  
Radboud University Nijmegen, The Netherlands  
Email: julien@cs.ru.nl

**Abstract**—We develop formal arguments about a bit clock synchronization mechanism for time-triggered hardware. The architecture is inspired by the FlexRay standard and described at the gate-level. The synchronization algorithm relies on a specific value of a counter. We prove or disprove values proposed in the literature. Our framework is based on a general and precise model of clock domain crossing, which considers metastability and clock imperfections. Our approach combines this model with the state transition representation of hardware. The result is a clear separation of analog and digital behaviors. Analog considerations are formalized in the logic of the Isabelle/HOL theorem prover. Digital arguments are discharged using the NuSMV model checker. To the best of our knowledge, this is the first verification effort tackling asynchronous transmissions at the gate-level.

## I. INTRODUCTION

Embedded systems comprise software applications, compilers, real-time operating systems, processors with memory management units and devices, as well as communication architectures. These different components form the layers of a stack. The top layer and the most abstract one is occupied by software applications. Going down in the abstraction, software applications together with an operating system are compiled into machine code and run on top of processing units and memories. Their gate level description constitutes the lowest layer of the stack. In the late 80's, Bevier *et al.* [2] demonstrated a first “stack proof”, *i.e.* a proof of a simulation theorem between the top layer and the bottom layer. The application of this approach to realistic embedded systems remains a challenge [12]. Computer systems are often *distributed*. One verified stack is not enough. One needs to prove correctness of stacks and their communications.

These communications are inherently asynchronous as in practice clocks of interconnected devices are not constant over time. This clock distortion induces possible metastable states of registers. The proof of *distributed stacks* requires the analysis of these phenomena at the gate-level. Moreover, in the context of realistic worst case execution time analysis, it is also necessary to know the duration of the transmission. Towards this end, a pencil and paper proof of an entire distributed systems was developed [1]. From this study, we formalized in

the logic of Isabelle/HOL [13] the bit transmission between independently clocked registers, assuming precise timing parameters and metastability [15].

The contribution of this paper is an important extension of these theoretical results and the definition and the application of a methodology for the verification of time-triggered hardware. We extend the Isabelle theory to allow for long-term jitter. This relieves the previous hypothesis about constant clock periods. The outcome is a general model of clock domain crossing. This model is combined with the semantics of transition systems used to describe hardware designs. This identifies constraints on the *digital design* that guarantee proper transmission assuming *analog behaviors*. These constraints can be solved by decision procedures. We use the integration in Isabelle of the model checker NuSMV [16]. We demonstrate this methodology on the verification of a time-triggered bus interface inspired by the FlexRay standard [5] and described at the gate-level. The statement of our theorem also includes the duration of the transmission. Our analysis identifies precisely the possible values of one crucial parameter of the bit clock synchronization mechanism. This proves and disproves values proposed in the literature.

The paper is structured as follows. In the next Section, we present our general model of clock domain crossing. We show in Section III how we use this model for hardware design verification. Section IV describes the time-triggered interface, which is verified in Section V. Related work is discussed in Section VI. Section VII concludes the paper.

## II. A FORMAL MODEL OF CLOCK DOMAIN CROSSING

### A. Signals and Clocks

Time is represented by the nonnegative reals ( $R_{\geq 0}$ ). A signal  $s$  is represented by a function  $s(t)$  from *real* time  $t$  to  $\{0, 1, \Omega\}$ : 1 and 0 mean “high” and “low” voltages;  $\Omega$  means any voltage.

The clock period of unit  $u$  is noted  $\tau_u$ . Periods are different from zero. The *date* of the  $c^{th}$  rising edge of clock  $clk_u$  of unit  $u$  is noted  $e_u(c)$ . It equals the product of  $c$  with the clock period:  $e_u(c) = c \cdot \tau_u$ .

Function  $e$  gives the ideal date of edges. In practice, it is impossible to guarantee constant clock periods. We assume that all clock periods of any clock deviate at most by a percentage  $\delta$  of a reference clock period. This reference clock

<sup>1</sup>Part of this work was carried out while the author was affiliated with the University of Saarland, Saarbrücken, Germany. This work was funded by the German Federal Ministry of Education and Research (bmb+f) in the framework of the Verisoft project under grant 01 IS C38.

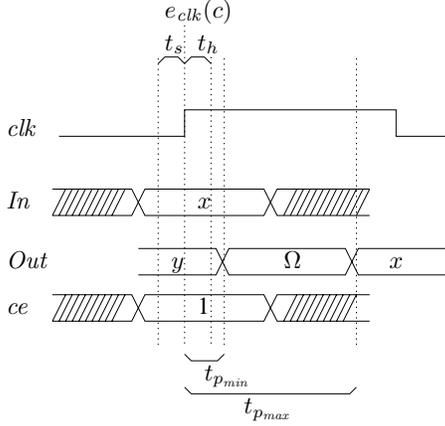


Fig. 1. Behavior of the register w.r.t clock edge  $c$

is named  $clk_{ref}$ . Its period is  $\tau_{ref}$ . Formally, we assume:

$$\Gamma_u \equiv 1 - \delta \leq \frac{\tau_u}{\tau_{ref}} \leq 1 + \delta \quad (1)$$

We are not interested in the deviation at each cycle, but in the number of cycles in which the number of ticks of two independent clocks may differ by at most one. Let  $\pi$  be that number. In this interval, the maximum drift between two clocks is obtained between the slowest and the fastest clocks allowed by Equation 1. Consequently, the ratio between the minimum and the maximum clock periods defines the lower bound of the drift. From Equation 1 and defining  $\pi = \frac{1-\delta}{2\delta}$ , we prove the following lemma:

**Lemma 1: Bounded Clock Drift.**

$$\Gamma_i \wedge \Gamma_j \rightarrow \frac{\pi}{\pi + 1} \leq \frac{\text{Min}(\tau_i, \tau_j)}{\text{Max}(\tau_i, \tau_j)}$$

This property is preserved for any number less than  $\pi$ .

## B. Analog Registers

Open intervals are represented using open squared brackets. Shifting an interval is noted  $x + [y : z]$  instead of  $[x + y : x + z]$ .

Registers consist of one input signal  $In$ , one clock signal  $clk$ , one control signal  $ce$  and one output signal  $Out$  (Fig. 1). A new value ( $x$ ) is input to the register at cycle  $c$ , which is defined by interval  $[e_u(c) : e_u(c + 1)]$ . During minimum propagation delay  $t_{pmin}$  the output signal equals previous value  $y$ . Because the control signal is high, the output oscillates (*i.e.* is  $\Omega$ ) before stabilizing at new value  $x$ . If the control signal is low, the output does not oscillate and keeps its old value  $y$ .

If the input or the control signals do not have a constant value during the *setup time* (noted  $t_s$ ) before edge  $c$  and during the *holding time* (noted  $t_h$ ) after edge  $c$ , the register may become metastable. This means that its output may still be  $\Omega$  after  $t_{pmax}$ . When this metastable state is resolved, the register reaches a defined value. Metastability cannot be avoided [9]. We assume that this resolution time is less than one clock period. Before giving our formal definition of analog registers, we define a few concepts.

$${}_aR_u(c, clk_u, ce_u, In_u, Out_u^0) \triangleq$$

```

if  $c = 0$  then  $\lambda t. Out_u^0$  else
  if  $\left\{ \begin{array}{l} \text{stade}p(e_u(c) - t_s, e_u(c) + t_h, ce_u) \\ \wedge \text{stade}p(e_u(c) - t_s, e_u(c) + t_h, In_u) \end{array} \right.$  then
    if  $ce_u(e_u(c)) = 1$  then
       $\lambda t. \left\{ \begin{array}{l} {}_aR_u(c - 1, \dots)(e_u(c)) : t \in e_u(c) + ]0 : t_{pmin}[ \\ \Omega : t \in e_u(c) + ]t_{pmin} : t_{pmax}[ \\ In_u(e_u(c)) : t \in e_u(c) + [t_{pmax} : \tau_u[ \\ \Omega : t \notin ]e_u(c) : e_u(c + 1)[ \end{array} \right.$ 
    else :: keep old value
       $\lambda t. \left\{ \begin{array}{l} {}_aR_u(c - 1, \dots)(e_u(c)) : \forall t \in ]e_u(c) : e_u(c + 1)[ \\ \Omega : t \notin ]e_u(c) : e_u(c + 1)[ \end{array} \right.$ 
    endif
  else :: metastability
     $\lambda t. \left\{ \begin{array}{l} {}_aR_u(c - 1, \dots)(e_u(c)) : t \in e_u(c) + ]0 : t_{pmin}[ \\ \Omega : t \in e_u(c) + ]t_{pmin} : \tau_u[ \\ x \in \{0, 1\} : t = e_u(c + 1) \\ \Omega : t \notin ]e_u(c) : e_u(c + 1)[ \end{array} \right.$ 
  endif
endif

```

Fig. 2. Definition of Analog Registers

The *metastability window* w.r.t. edge  $c$  of register  $u$  (noted  $MW_u^c$ ) is defined by interval  $e_u(c) + [-t_s : t_h]$ .

A signal  $s$  is stable during time interval  $[t_1 : t_2]$  if it holds the value at time  $t_1$  until time  $t_2$ . A signal  $s$  has a defined value during time interval  $[t_1 : t_2]$  if it never equals  $\Omega$  during that interval. Formally, this is expressed as follows<sup>1</sup>:

$$\text{stade}p(t_1, t_2, s) \triangleq \exists b \in \{0, 1\}, \forall t \in [t_1 : t_2], s(t) = b$$

The formal definition of the analog behavior is given by function  ${}_aR_u$  (Fig 2). We are interested in the output value of a register for all real times during cycle  $c$ . Function  ${}_aR_u$  takes as arguments a cycle  $c$ , a clock signal  $clk_u$ , a clock enable control signal  $ce_u$ , an input signal  $In_u$ , and the initial output value  $Out_u^0$ . It generates a signal.

If no setup or holding time violation occurs, the register behaves normally. If the control signal is low, the register keeps its old value (at the previous cycle  $c - 1$ ); if the control signal is high the output keeps its previous value during  $t_{pmin}$ , then oscillates (*i.e.* is  $\Omega$ ) to finally reach its final value at time  $e_u(c) + t_{pmax}$ . If input signal  $In_u$  or control signal  $ce_u$  is not stable and defined during the metastability window, the register becomes metastable. The output equals the previous computation until  $t_{pmin}$  (included) and  $\Omega$  afterwards. At the end of the cycle, metastability has been resolved and the output equals an arbitrary but defined value. To make the function total,  $\Omega$  is output for all times outside the cycle. To alleviate our notation, we shall write  ${}_aR_u^c$  instead of  ${}_aR_u(c, clk_u, ce_u, In_u, Out_u^0)$ .

Formally, all timing parameters ( $t_h, t_s, t_{pmin}, t_{pmax}$ ) are percentages. We assume that their sum is less than 1. Their value depends on the local clock period. In the remainder of this

<sup>1</sup>Note: *stade*p means **stable, defined, predicate**

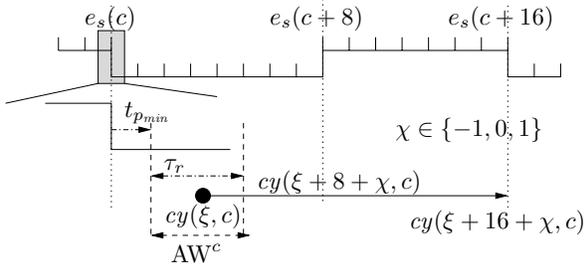


Fig. 3. Relating Receivers and Senders

paper, propagation delays are relative to the sender clock and setup and holding times are relative to the receiver clock period.

### C. Relating Receivers to Senders

The relation between a sender and a receiver is pictured in Fig. 3. A sender starts sending three different bits at edge  $\#c$ ,  $\#c+8$ , and  $\#c+16$ . If we take a closer look around edge  $c$ , the sender output is not modified before  $e_s(c) + t_{pmin}$ , when it moves from  $y$  to  $\Omega$  (see Fig 1 for more details). If a receiver samples before that date, it will get the old value. In contrast, sampling *strictly after* that date will affect the receiver, either it will get metastable, or it will detect a new value. Let  $\xi$  be the first receiver edge after  $e_u(c) + t_{pmin}$ . As this edge is the first one to be affected by the behavior of the sender, we denote it as “marked with edge  $c$ ”, noted  $cy(\xi, c)$ . If there is no ambiguity, we may drop the first argument. Edge  $\xi$  occurs in time interval  $e_s(c) + t_{pmin} + ]0 : \tau_r]$ . This interval defines the “affected window w.r.t. edge  $c$ ”, noted  $AW^c$ . Formally, we have the following definition:

**Definition 1: Affected Cycle.**  $cy(\xi, c) \equiv e_r(\xi) + t_h \in AW^c$

Let us suppose that edge  $\xi$  is in  $AW^c$ . If the sender sends another bit within a number of cycles ( $\alpha$ ) less than our bound  $\pi$ , the corresponding affected window may be seen by the receiver with a potential error of one cycle, *i.e.* at  $e_r(\xi + \alpha \pm 1)$ . This means that subsequent marks are known with the same error. Fig 3 shows these marks for  $\alpha = 8$  and  $\alpha = 16$ .

Formally, we have the following theorem:

**Theorem 1: Regular Affected Windows**

$$\begin{aligned} & \Gamma_{clk_r} \wedge \Gamma_{clk_s} \wedge 0 < \alpha \leq \pi \wedge cy(\xi, c) \\ & \rightarrow \bigvee_{\chi \in \{-1, 0, 1\}} e_r(\xi + \alpha + \chi) \in AW^{c+\alpha} \end{aligned}$$

**Proof.** We do a case analysis depending on the position of  $\xi + \alpha$  regarding the affected window  $AW^{c+\alpha}$ . If  $e_r(\xi + \alpha)$  is (1) before  $AW^{c+\alpha}$ , we prove  $e_r(\xi + \alpha + 1) \in AW^{c+\alpha}$ ; (2) within  $AW^{c+\alpha}$ , this proves the obvious case where  $\chi = 0$ ; (3) after  $AW^{c+\alpha}$ , we prove  $e_r(\xi + \alpha - 1) \in AW^{c+\alpha}$ .  $\square$

### D. Safe Sampling Window

In case of metastability, we always assume resolution to the negation of the expected input. Thus, an extra delay may be introduced. This is represented by the *metastability factor* ( $\beta$ ). Metastability can only happen if the affected cycle (minus the setup time) appears while the sender output is undefined,

*i.e.* before  $e_s(c) + t_{pmax}$ . In this case, the metastability factor returns 1. It returns 0 otherwise. Formally, the *metastability factor* is a function, which takes as arguments cycles  $\xi$  and  $c$ , and two clocks.

**Definition 2: Metastability Factor.**

$$\beta(\xi, c, clk_s, clk_r) \triangleq$$

**if**  $e_r(\xi) - t_s \leq e_s(c) + t_{pmax}$  **then** 1 **else** 0

To alleviate the notation, we shall write  $\beta_c^\xi$  instead of  $\beta(\xi, c, clk_s, clk_r)$ .

To ensure that the receiver will not always sample  $\Omega$ 's, the sender keeps its output constant for several cycles (say  $k$  cycles). Consequently, there is only one metastability window and if  $k$  is big enough there exists a “sweet spot” in which the receiver can sample safely. Formally, the safe sampling window of length  $k$  w.r.t. cycle  $c$  (noted  $SSW_k^c$ ) is denoted by interval  $[e_u(c) + t_{pmax} : e_u(c + k + 1) + t_{pmin}]$ .

We prove that under our drift hypothesis,  $SSW_k^c$  entails up to  $k - 1$  receiver cycles (or  $k$  edges), even in case of metastability.

**Theorem 2: SSW's are large enough.**

$$\begin{aligned} & \Gamma_{clk_r} \wedge \Gamma_{clk_s} \wedge cy(\xi, c) \wedge n + 1 \leq k \leq \pi \\ & \rightarrow \forall l \leq n, cy(\xi + \beta_c^\xi + l) + [-t_s : t_h] \in SSW_k^c \end{aligned}$$

### E. Clock Domain Crossing Correctness

Our main theorem proves that sampling in a safe sampling windows is correct. We assume that the sender creates a safe sampling window of length  $k$ , control and input bits must be stable and defined during all sender metastability windows, clock drift is bounded. We assume that cycle  $\xi$  is in  $SSW_k^c$ .

**Theorem 3: Correct Transfer.**

$$\begin{aligned} & \Gamma_r \wedge \Gamma_s \wedge cy(\xi, c) \text{ (*bounded drift, affected cycle*)} \\ & \wedge \text{ (*SSW}_k^c \text{ *)} \\ & \wedge ce_s(e_s(c)) = 1 \wedge \forall l \in [1 : k], ce_s(e_s(c + l)) = 0 \\ & \wedge c > 0 \wedge n + 1 \leq k \leq \pi \\ & \wedge \forall l \in [0 : k + 1], \text{ (*input *)} \\ & \quad \text{stadep}(e_s(c + l) - t_s, e_s(c + l) + t_h, In_s) \\ & \wedge \forall l \in [0 : k + 1], \text{ (*control*)} \\ & \quad \text{stadep}(e_s(c + l) - t_s, e_s(c + l) + t_h, ce_s)) \\ & \wedge \text{ (*analog connection*)} \\ & \quad \forall c, In_r = {}_aR_s(c, clk_s, ce_s, In_s, Out_s^0) \wedge \forall t, ce_r(t) = 1 \\ & \wedge e_r(\xi) + [-t_s : t_h] \in SSW_k^c \text{ (* good cycle *)} \\ & \rightarrow {}_aR_r^\xi(e_r(\xi + 1)) = In_s(e_s(c)) \end{aligned}$$

**Proof.** First, Theorem 2 gives us the position of receiver edges in the safe sampling window. Then, we case split on the position of the metastability window around cycle  $\xi$ . We set two reference points:  $e_s(c + 1)$  and  $e_s(c + 1 + k)$ . We prove the conclusion for 5 cases depending on the position of the metastability window regarding these points.  $\square$

## III. ANALOG TRANSFER IN A DIGITAL WORLD

Our model of clock domain crossing mentions analog entities only. The semantics is based on functions and a dense representation of time. Ultimately, we want to use this model to verify hardware designs described in another semantics, which is based on a discrete notion of time and transition

functions. Before describing our approach, we define type conversion functions and rephrase Theorem 3 to match bits and not signals.

#### A. Type Conversions

The conversion from bit lists to signals is done by function  $\gamma$ . We do not give a particular definition to this function. We only assume that it produces a signal such that during the metastability window around cycle  $i + 1$ , it outputs the value with index  $i$  in the bit list. This property is defined by predicate  $bv2sp$ :

$$bv2sp(\gamma, l_u) \triangleq \forall t, i, t \in MW_u^{i+1} \rightarrow \gamma(l_u) = l_u[i]$$

The conversion from signals to bits is done by function  $\zeta$ , which takes as input a signal and a time. If the value of the signal at that time is a bit value, then this value is returned. Otherwise, *some* bit value is returned.

$$\zeta(s, t) \triangleq \text{if } s(t) \in \{0, 1\} \text{ then } s(t) \text{ else } x \in \{0, 1\}$$

#### B. Transfer Correctness in the Digital World

Let lists  $ce_s$  and  $In_s$  be the bit lists containing values given to the analog sender register. If they both satisfy predicate  $bv2sp$ , list element  $ce_s[c - 1]$  or  $In_s[c - 1]$  corresponds to the bit value given to the sender analog register at time  $e_s(c)$ .

Theorem 3 is embedded into a digital context in the following statement. We assume that (a) clock drift is bounded; (b) function  $\gamma$  correctly translates bit lists  $ce_s$  and  $In_s$ ; (c) the digital control bits are high once and then low  $k$  times. Analog hypotheses are concerned with the connection of the sender with the receiver and the clock drift. Obviously, they cannot be “digitalized”. Under these assumptions, we prove that the “digitalized” output of the **analog** receiver register equals the **digital** input of the sender at cycle  $c$ . In the remainder of this paper, we will denote the conjunction of the hypotheses of this theorem by  $\mathcal{H}$ .

#### Theorem 4: Back to the Digital World.

$$\begin{aligned} & \Gamma_r \wedge \Gamma_s \wedge n + 1 \leq k \leq \pi \wedge cy(\xi, c) \\ & \quad (*\text{bounded drift, cy}(c)*) \\ \wedge & \quad \forall c, In_r = {}_aR_r(c, clk_s, \gamma(ce_s), \gamma(In_s), Out_s^0) \\ \wedge & \quad \forall t, ce_r(t) = 1 \quad (*\text{analog link}*) \\ \wedge & \quad bv2sp(\gamma, ce_s, clk_s) \wedge bv2sp(\gamma, In_s, clk_s) \\ & \quad (*\text{modeling hypotheses}*) \\ \wedge & \quad ce_s[c + \alpha - 1] = 1 \wedge c > 0 \\ \wedge & \quad \forall l \in [1 : k], ce_s[c + l - 1] = 0 \quad (*\text{sender OK}*) \\ \wedge & \quad e_r(\xi) + [-t_s : t_h] \in SSW_k^c \quad (*\text{good cycle}*) \\ \rightarrow & \quad \zeta({}_aR_r^c, e_r(\xi + 1)) = In_s[c - 1] \end{aligned}$$

**Proof.** By definition of predicate  $bv2sp$ ,  $\gamma(In_s)$  and  $\gamma(ce_s)$  are *stade*p for the required cycles. Theorem 3 concludes.  $\square$

#### C. Principle and Soundness

Fig. 4 illustrates our integration of our analog results in the analysis of digital designs. Our clock domain crossing model (CDC) is shown inside the dashed box. The remainder of the figure corresponds to digital designs that are actually used

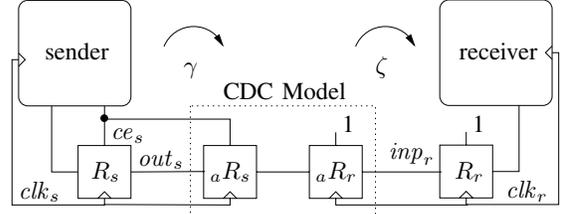


Fig. 4. Mixing Analog and Digital Signals

to synthesize hardware. These designs are not modified. Our model is simply inserted as a filter of the receiver inputs.

Digital designs are represented by their transition function, one application of which represents the computation of one clock cycle. The sender and the receiver parts are analyzed separately. The analysis of the sender does not need any analog arguments. It mainly consists of the proof that sender output  $out_s$  follows a specific frame format. The analysis of the receiver is done assuming correctness of the sender and that the connection of receiver input  $inp_r$  is done through our CDC model. We write that an element  $s_u$  of unit  $u$  has bit-value  $x$  at cycle  $c$  - *i.e.* after  $c$  applications of the transition function - as  $s_u^c = x$ . Formally, we assume that the value of input bit  $inp_r$  at hardware cycle  $c$  equals the output value of register  ${}_aR_r$  at the date of edge  $c + 1$ :

$$\forall c, inp_r^c = \zeta({}_aR_r^c, e_r(c + 1)) \quad (2)$$

The left hand side represents the value that should be in register  $R_r$  at  $c + 1$ . As the analog register is not part of the transition function of the receiver, one application of the latter compensates this difference. The right hand side is always a defined value. This Equation only holds when  $R_r$  is not metastable. As discussed in the next sub-section, it is always the case when we use Equation 2.

#### D. Proof Method

Our proof method uses Theorems 2, 1, 4 and Equation 2. We also need a mark  $cy(\xi, c)$  which connects receiver cycle  $\xi$  with the beginning of a safe sampling window started at sender cycle  $c$ . From this mark, we obtain from Theorem 2 that there are  $n + 1$  receiver edges in the safe sampling window. These edges may be shifted by one cycle depending on the resolution of metastable states. Then, we obtain from Theorem 4 that register  ${}_aR_r$  outputs  $out_s^c$  at the date of these edges. Because we are outside metastable behaviors, we obtain from Equation 2 inputs for the receiver. Once these inputs are known, the analysis is back to the digital world and decision procedures apply. We obtain subsequent inputs using a similar reasoning and the marks given by Theorem 1.

## IV. TIME-TRIGGERED BUS INTERFACES

We present an implementation of a time-triggered bus interface inspired by the FlexRay standard [5] for safety-critical automotive applications. This design has already been

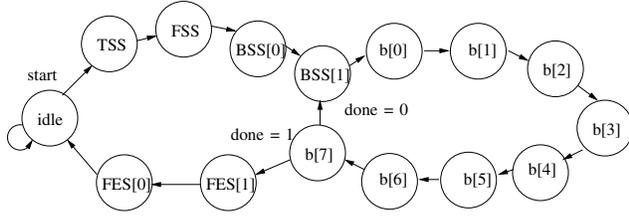


Fig. 5. Control Automaton

presented [1]. It can be translated to Verilog [14] and synthesized on FPGA. It will be available at the Verisoft Repository<sup>2</sup>.

### A. Protocol Overview

We consider an arbitrary number of units connected through a shared bus. Each unit can send and receive messages. Idle units put one's on the bus. Let TSS = 0 be the transmission start sequence, FSS = 01 be the frame start sequence, BSS = 10 be the byte start sequence and FES = 01 be the frame end sequence. Let  $\langle a, b \rangle$  be the concatenation of bit vectors  $a$  with  $b$ . A message  $m$  of  $l$  bytes is encapsulated into a frame  $f(m)$  with the following format:

$$f(m) = \langle \text{TSS}, \text{FSS}, \text{BSS}, m[0], \dots, \text{BSS}, m[l-1], \text{FES} \rangle$$

Each bit of a frame is sent 8 times.

### B. Sender Module

The sender implements the protocol by the control automaton in Fig. 5. As specified by the protocol, in each state the corresponding bit is generated 8 times. The sender is connected with the shared bus through a register named  $R_s$  with control enable bit  $ce_s$ . This paper focuses on the verification of message reception. We do not detail the sender implementation any further.

### C. Receiver Implementation: Bit Clock Synchronization

The receiver module implements the same state-automaton as the sender. In each state, the receiver is expecting to receive the corresponding bit of the frame. Beside the automaton, the relevant part of this receiver consists of the input stage pictured in Fig. 6. The first two registers form a "synchronizer" used to remedy to metastability. A five majority vote is performed. Signal  $sync$  is used to detect the synchronization sequence BSS. It is high if and only if the current voted bit does not equal its previous value and the state automaton is either in state *idle* or in state BSS[1]. When  $sync$  is high counter  $cnt$  is reset to 000 in the next cycle. The state automaton is clocked by signal  $strobe$ , which is high each time the counter reaches value 010 and the automaton is not synchronizing, *i.e.* when signal  $sync$  is low. Each time  $strobe$  is high, the voted bit is stored in shift register BYTE. When the last bit has been stored (*i.e.* automaton is in state b[7]) and signal  $strobe$  is high, signal  $rb.we$  turns high and BYTE is written to the main receiver buffer.

<sup>2</sup><http://www.verisoft.de/VerisoftRepository.html>

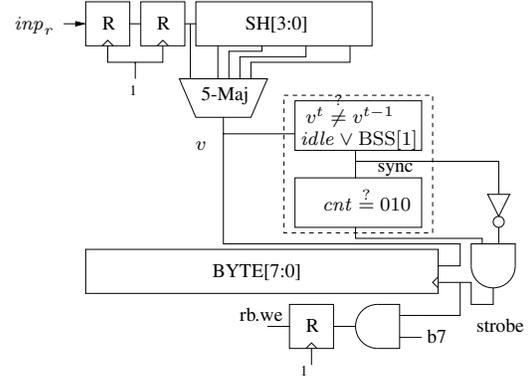


Fig. 6. Input Stage

This implementation uses the synchronization mechanism described in the FlexRay standard. Synchronization is performed by resetting the counter when receiving the BSS sequence. Our implementation differs slightly from the FlexRay guidelines. The standard suggests to reset the counter to 010 and to strobe when it reaches 101. We reset to 000 and strobe at 010. So, we strobe one cycle earlier. In [1], the counter is reset to 000 and *strobe* is high when  $cnt$  is 100.

## V. FORMAL VERIFICATION

### A. Sender Correctness

The sender is proven to effectively generate each bit 8 times. This discharges the digital hypotheses of Theorem 4. Formally, this is defined as follows:

**Definition 3: Correctness of  $ce_s$ .**

$$\text{WF}_{ce}(ce_s, L, k, c) \equiv \forall i < L, ce_s[c + 8 \cdot i] = 0 \\ \wedge \forall j \in [1 : k], ce_s[c + 8 \cdot i + j] = 0$$

We prove that the sender generates frames with the specified format. For the purpose of this paper, we are only concerned with synchronization bits, *i.e.* the BSS sequence. This is expressed by the following predicate:

**Definition 4: Partial Correctness of  $In_s$ .**

$$\text{WF}_{In}(In_s, L, c) \equiv \forall i < L, \forall y \in [0 : 7] \\ \left\{ \begin{array}{l} In_s[c + 80 \cdot i + 16 + y - 1] = 1 \\ \wedge In_s[c + 80 \cdot i + 24 + y - 1] = 0 \end{array} \right.$$

### B. Receiver Correctness Statement

The correctness of a time-triggered interface is achieved if for the transmission of any byte of a frame there exists a hardware cycle from which the interface recovers that byte. This requires the proof that (1) depending on the position of the BSS[0]-mark ( $cy(\text{BSS}[0])$ ) the state automaton strobes the right voted bits, and (2) this happens soon enough to match the sender output. The first statement expresses that the automaton indeed synchronizes and the second that this synchronization is good enough to sample properly. The final proof uses these two statements to prove by induction over the number of bytes that the whole frame is recovered. We assume that the state automaton is initially in state "idle" and that the first mark

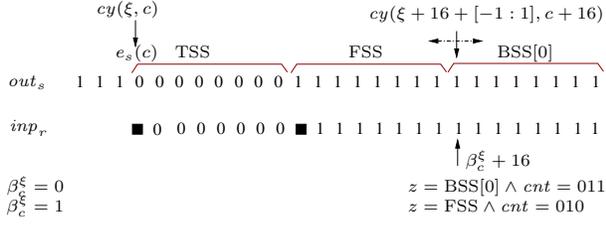


Fig. 7. Initial Transmission Phase

is known. The final statement is Theorem 5 below. The first line of the conclusion states that for all bytes we detect a mark for BSS[0]. The second line states that at this cycle the control automaton could be in different states with different values of its counter. This different values come from previous bytes which may have suffered from clock drifts. The last line state that under this uncertainty the automaton samples byte  $\#i$  properly. This theorem also proves lower and upper bounds on the time at which the last byte is recovered. Using the mark of the conclusion, these bounds can be expressed as functions of the reference clock and the time ( $e_s(c)$ ) when the first bit is put on the bus by the sender.

**Theorem 5: Transmission Correctness.**

$$\begin{aligned}
& \mathcal{H} \wedge z^\xi = \text{idle} \wedge cy(\xi, c) \\
& \forall c, \text{inp}_r^c = \zeta({}_a R_r^c, e_r(c+1)) (* \text{ A/D mix} *) \\
& \text{WF}_{ce}(ce_s, L, k, c) \wedge \text{WF}_{In}(out_s, L, c) \\
\rightarrow \\
& \forall i < l, \exists \nu, cy(\nu, c + 16 + 80 \cdot i) (* \text{ mark} *) \\
& \wedge z^\nu = (\text{FSS} \vee b[7]) \wedge cnt^\nu = (001 \vee 010) \\
& \vee z^\nu = \text{BSS}[0] \wedge cnt^\nu = (011 \vee 100) \\
& \wedge z^{\nu+78+w} = \text{BSS}[0] \wedge cnt^{\nu+78+w} = 010 \\
& \wedge \text{BYTE}^{\nu+79+w} = \langle out_s^{c+16+80 \cdot i+8 \cdot (j+2)} \rangle
\end{aligned}$$

where  $w \in [0 : 3]$  and  $j \in [7 : 0]$

The proof is done by induction over  $i$ . For space reason, we only detail the base case, which is pictured in Fig. 7. The first two lines show the output of the sender and how it is seen by the receiver. Black boxes indicate possible metastability.

Because of clock drift, the BSS[0]-mark may appear on the receiver side 15, 16 or 17 cycles after  $\xi$ . There is a potential metastability at cycle  $\xi$ . Depending on the value reached after resolution – that is depending on the value of  $\beta_c^\xi$  – the receiver automaton reaches different state and counter values when the BSS[0]-mark is detected. In the figure, we show these values at  $\beta_c^\xi + 16$ , where the automaton is either in state BSS[0] with a counter at 011 or in state FSS with a counter at 010. In the following sections, we prove that the receiver recovers a byte for all these possibilities.

### C. Traversing Synchronization Edges

Our reasoning is illustrated in Fig. 8. The first two lines show the output of the sender and how it is seen by the receiver. Black boxes indicate possible metastability. Question marks are used to denote unknown values.

We fix the initial step of the lemma to match the date of the detection of the BSS[0]-mark. We consider the case where

the receiver is in state BSS[0] with a counter value at either 011 or 100.

According to Theorem 1 and assuming that the BSS[0]-mark is known, the BSS[1]-mark has three possible dates. The potential metastability around that edge has the same three dates. We consider bits sampled by the receiver at these dates unknown. Another source of uncertainty resides in factor  $\beta$ . It is already represented by metastability. Therefore, at most three bits are unknown. Depending on the values of these three bits, the automaton will spend more or less time in the states of BSS. There is synchronization if the lower and the upper bound on this number of cycles allow proper sampling. This bounds are defined by the next lemma which imposes that the automaton reaches state  $b[0]$  with counter value 011 in at least 15, and at most 18 cycles.

Let  $t$  be the date of the affected cycle of BSS[0]. If the three unknown bits are 0 (see line 3 in Fig. 8), signal *sync* is high at  $t + 7 + 4 = t + 11$ . The counter is reset, and signal *strobe* is high at  $t + 11 + 3 = t + 14$ . In the next cycle, the automaton reaches state  $z^{t+15} = b[0]$ . For any lower value of the counter, the automaton will reach this state earlier.

If the unknown bits are 1, signal *sync* is high at  $t + 10 + 4 = t + 14$ . If the counter was 100 initially, then it has reached value 010 and *strobe* is high. At the same time, signal *sync* is high, the automaton stays in BSS[0] and the counter is reset. At cycle  $t + 17$ , *strobe* is high and the automaton reaches  $b[0]$  with a correct counter value at  $t + 18$ . For any larger value, the automaton requires more cycles to reach this state.

From Theorem 4 and considering the possible values of  $\beta_c^\xi$ , we know for sure 6 bits of BSS[0] (from  $t + 1$  to  $t + 6$ ) and 6 bits of BSS[1] (from  $t + 9$  to  $t + 14$ ). Assuming that only these input values are known, the rest of the proof is purely digital. It is expressed by the following lemma, the proof of which is fully automatic.

**Lemma 2: From BSS[0] to BSS[1].**

$$\begin{aligned}
& \forall u \in [0 : 6], \text{inp}^{t+1+u} = 1 (* \text{ 6 bits of BSS}[1] *) \\
& \wedge \forall v \in [0 : 6], \text{inp}^{t+9+v} = 0 (* \text{ 6 bits of BSS}[0] *) \\
& \wedge ((z^t = \text{BSS}[0] \wedge cnt^t = 010 \vee 011) (* \text{ states and} *) \\
& \vee (z^t = \text{FSS} \wedge cnt^t = 001 \vee 010)) (* \text{ misalignment} *) \\
\rightarrow & \bigvee_{w \in [0:3]} z^{t+15+w} = b[0] \wedge cnt^{t+15+w} = 011
\end{aligned}$$

**Proof.** By NuSMV.  $\square$

### D. Strobing Correct Bits

The next lemma states that whatever happens in the traversal of “synchronization” edges, strobe points always hit correct voted bits. We consider hypotheses similar to the previous lemma. The BSS[0]-mark matches the start point of the lemma. The automaton could be in state BSS[0] with counter at 011 or 100, or in state FSS with counter at 001 or 010. The reasoning is illustrated in the right part of Fig. 8, where voted bits are shown instead of the input.

Formally, we prove that register BYTE contains the correct frame 79 to 82 cycles following the first bit of the synchronization sequence. The proof shows the exact values of the counter that allow proper transmission.



## VI. RELATED WORK

The first verification effort about physical layer protocols was carried out by Moore [11]. Moore developed a general model of asynchronous communications as a function in the logic of the ACL2 theorem prover [8]. Moore's model assumes distortion around sampling edges and do not allow for clock jitter. Sender and receiver modules are also represented by two functions. Moore's correctness criteria states that the composition of these three functions is an identity. He applied this approach to the verification of a Biphase-Mark protocol.

Moore's work inspired many studies around this protocol. Recently, Vaandrager and de Groot [17] modeled the protocol and analog behaviors using a network of timed-automata. Their model is slightly more general than Moore's and allows for clock jitter. They can derive tighter bounds for the Biphase-Mark protocol. Previously, timed-automata have been used to verify a low level protocol based on Manchester encoding and developed by Philips [3]. Another recent proof of the Biphase-Mark protocol has been proposed by Brown and Pike [4]. They developed a general model of asynchronous communications in the formalism of the tool SAL [10] developed at SRI. Their model includes clock jitter and metastability. Using  $k$ -induction, the verification of the parameterized specification of Brown and Pike is largely automatic. All these studies tackle *protocol specification* only. They prove functional correctness. We prove a more precise theorem about a gate-level hardware implementation and from which bounds on the transmission duration can be derived.

Regarding hardware verification, Hanna [6], [7] used predicates to approximate analog behaviors at the transistor level. The predicates can be embedded in digital proofs. His work is not specifically targeted to communication circuits and does not consider timing parameters, metastability or clock drift. We consider only gates and not their structure in terms of transistors.

## VII. CONCLUSION AND FUTURE WORK

Reliable transmission between two independent clocked devices is performed using bit clock synchronization, which is achieved by resetting a counter when detecting a synchronization sequence. This specific value is a crucial parameter. We have developed a general and precise model of asynchronous communications and defined a methodology to use this model for hardware design verification. We have proven the exact possible values for this parameter. This proves and disproves values proposed in the literature.

The model of clock domain crossing is about 2,000 lines and is available on the web<sup>3</sup>. The proof presented here was developed in about one man-year and is about 8,000 lines. Most of it is dedicated to the deduction of valid digital inputs from the analog transmission. This technique is independent of the design under verification. The analysis of similar designs will mainly amount to re-prove all digital lemmas.

The case study is extracted from a more complex design which includes a scheduler implementing a high-level clock synchronization algorithm. We are currently applying our approach to the verification of this component, moving towards a fully verified distributed system.

## ACKNOWLEDGMENTS

Tsverdyshev provided a precious technical support on the use of NuSMV within Isabelle. Shadrin wrote the translator from Isabelle to Verilog. This work initiates from the lecture "Computer Architecture 2 – Automotive Systems" given by Paul at Saarland University and notes taken by students<sup>4</sup>.

## REFERENCES

- [1] S. Beyer *et al.*, Towards the Formal Verification of Lower System Layers in Automotive Systems, In *Proc. of the 23rd IEEE International Conference on Computer Design (ICCD 2005)*, 2005.
- [2] W. R. Bevier, W. A. Hunt Jr., J Strother Moore and W. D. Young, An Approach to Systems Verification, *Journal of Automated Reasoning* 5(4):411-428, 1989.
- [3] D. Bosscher, I. Polak and F. Vaandrager, Verification of an Audio Control Protocol, In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, pp. 170-192, LNCS 863, Springer 1994
- [4] G.M. Brown and L. Pike, Easy Parameterized Verification of Biphase Mark and 8N1 Protocols, In *Proc. of 12th Intl. Conf. on Tools and the Construction of Algorithms (TACAS'06)*, pp. 58-72, LNCS 3920, Springer, 2006.
- [5] FlexRay Communication System – Protocol Layer Specification v2.1, Rev A, FlexRay Consortium, December 2005.
- [6] K. Hanna, Reasoning About Real Circuits, in *Proc. of the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, pp 235-253, Springer-Verlag, 1994
- [7] K. Hanna, Automatic Verification of Mixed-Level Logic Circuits, In *Proc. of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD '98)*, LNCS 1522, pp 133-166, Springer-Verlag 1998.
- [8] M.Kaufmann, P.Manolios, and J.Moore, *Computer Aided Reasoning: an Approach*. Kluwer Academic Press, 2002.
- [9] R. Männer, Metastable States in Asynchronous Digital Systems: Avoidable or Unavoidable ? *Microelectronics Reliability*, 28(2):295-307, 1988
- [10] L. de Moura *et al.*, SAL 2, In *Computer-Aided Verification (CAV'04)*, pp. 496-500, LNCS 3114, Springer, 2004.
- [11] J Strother Moore, A Formal Model of Asynchronous Communications and Its Use in Mechanically Verifying a Biphase Mark Protocol, *Formal Aspects of Computing*,6(1) 60-91, 1993.
- [12] J Strother Moore, A Grand Challenge Proposal for Formal Methods: A Verified Stack, In *10<sup>th</sup> Anniversary Colloquium of UNU/IIST*, B. K. Aichernig and T. S. E. Maibaum editors, LNCS 2757, pp 161-171, Springer, 2003.
- [13] T. Nipkow, L.C. Paulson and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.
- [14] V. Sagdeo, *The Complete Verilog Book*, Springer, 1998
- [15] J. Schmaltz, A Formal Model of Lower System Layer, In *Proc. of Formal Methods in Computer-Aided Design (FMCAD'06)*, San Jose, CA, USA, November 12-16, pp 191-192, IEEE/ACM, 2006.
- [16] S. Tverdyshev, Combination of Isabelle/HOL with Automatic Tools. In *Proc. of 5th International Workshop on Frontiers of Combining Systems, FroCoS 2005*, LNCS 3717, pp 302-309, Vienna, Austria, September 19-21, 2005.
- [17] F.W. Vaandrager and A.L. de Groot, Analysis of a Biphase Mark Protocol with UPAAL and PVS, *Formal Aspects of Computing* 18(4):433-458, Springer, 2006.

<sup>3</sup>[www.cs.ru.nl/~julien/](http://www.cs.ru.nl/~julien/), then Research

<sup>4</sup>[www-wjp.cs.uni-sb.de/lehre/lehre.php](http://www-wjp.cs.uni-sb.de/lehre/lehre.php)

# Modeling Time-Triggered Protocols and Verifying Their Real-Time Schedules

Lee Pike  
Galois, Inc.  
leepike@galois.com

**Abstract**—*Time-triggered systems* are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. Time-triggered protocols depend on the synchrony assumption the underlying system provides, and the protocols are often formally verified in an untimed or synchronous model based on this assumption. An untimed model is simpler than a real-time model, but it abstracts away timing assumptions that must hold for the model to be valid. In the first part of this paper, we extend previous work by Rushby [1] to prove, using mechanical theorem-proving, that for an arbitrary time-triggered protocol, its real-time implementation satisfies its untimed specification. The second part of this paper shows how the combination of a bounded model-checker and a satisfiability modulo theories (SMT) solver can be used to prove that the timing characteristics of a hardware realization of a protocol satisfy the assumptions of the time-triggered model. The upshot is a formally-verified connection between the untimed specification and the hardware realization of a time-triggered protocol with respect to its timing parameters.

## I. INTRODUCTION

Digital control systems are being designed for use in safety-critical contexts such as automobiles (“drive-by-wire”) and commercial aircraft (“fly-by-wire”) [2], [3]. Safety-critical systems embedded in commercial aircraft must have a failure rate probability no greater than  $10^{-9}$  per hour of operation [4], [5]. A design error causing a system to have a higher rate of failure—say a failure rate of  $10^{-8}$  per hour—is unacceptable, yet it is infeasible to determine whether a system has this reliability through testing alone [6]. The inability to demonstrate correctness through testing motivates us to *prove* these systems are correct.

The specific class of control systems considered in this paper are *time-triggered systems*. Time-triggered systems are implemented as distributed systems in which each node in the system is independently-clocked, and under normal operating conditions, synchronization mechanisms maintain tight synchronization among the local clocks [5]. When the nodes are tightly synchronized, the temporal behavior of the system can be abstracted as if the nodes execute in lock-step. This sort of abstraction is characterized by the *synchronous model* or *untimed model*. At the level of abstraction that the synchronous model provides, formal correctness proofs of the protocols are difficult but feasible [7], [8].

The synchronous abstraction depends on a *realization* (i.e., a concrete implementation—hardware and/or software executing on hardware) satisfying key properties regarding scheduling, message delays, clock skew, message-reception windows, and

so forth. A more fine-grained model that addresses these aspects for time-triggered systems is the *time-triggered model*. The essential feature of this model, as opposed to an asynchronous model, is that while the local clocks of individual nodes may not be perfectly synchronized, their disharmony is bounded.

As demonstrated by Rushby, a subset of time-triggered protocols can be systematically shown to implement their synchronous specifications, provided certain timing constraints are met by the underlying system [1]. However, Rushby’s work suffers two shortcomings. First, despite his formal verification of the time-triggered model in the PVS mechanical theorem-prover [9], three of the four system assumptions (or formally, axioms) he postulates not only fail to model the actual behavior of time-triggered systems, but are in fact inconsistent. In a recent note by this author, these axioms were mended and their consistency proved [10]; we use the mended axioms herein. Second, the model is too constrained to model some actual realizations of time-triggered protocols. Therefore, we generalize the theory to accommodate time-triggered protocol realizations (and their optimizations) that fall outside the theory developed. Specifically, the theory is extended to accommodate (1) event-triggered behavior, (2) communication delays, (3) reception windows, (4) non-static clock skew, and (5) pipelined rounds (these generalizations are explained and justified in Section III). These extensions are used to model, for example, the time-triggered protocol implementations of NASA Langley’s SPIDER, an ultra fault-tolerant fly-by-wire communications bus [11]. This generalized time-triggered model, a proof of its consistency, and a proof that an arbitrary time-triggered protocol implements its synchronous model have been formulated in PVS (based on Rushby’s original PVS specifications and proofs), and those specifications and proofs are available online.<sup>1</sup>

In the second half of this paper, we demonstrate how to formally verify that a protocol realized in hardware satisfies the scheduling constraints of the time-triggered model. The verification is done using SRI’s SAL family of model-checkers, which combines a bounded model-checker implementing  $k$ -induction with SRI’s satisfiability modulo theories (SMT) solver, Yices, to prove LTL safety properties over infinite-

<sup>1</sup>Specification and proof files can be found at [http://www.cs.indiana.edu/~leepike/pub\\_pages/fmcd.html](http://www.cs.indiana.edu/~leepike/pub_pages/fmcd.html). To improve the presentation in this paper, slight syntactic modifications are made from the PVS specifications (e.g., some functions are uncurried).

state systems (our proofs essentially depend on the theory of linear arithmetic and uninterpreted functions) [12]. These specifications and proofs can also be found on-line.<sup>1</sup> Besides being a novel application of formal verification in general, the approach showcases a particularly successful application of recently-developed infinite-state model-checking techniques.

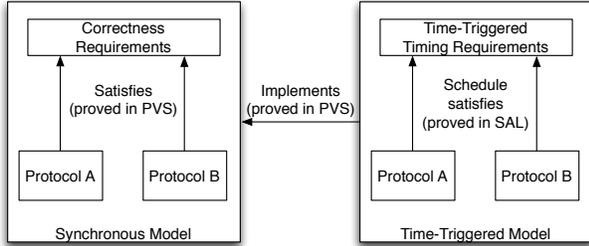


Fig. 1. Time-Triggered Protocol Verification Strategy

Taken together, we have a methodology for proving that a hardware realization of a time-triggered protocol implements its synchronous specification, with respect to its timing parameters, as illustrated in Figure 1.

The remainder of this paper proceeds as follows. The synchronous model’s syntax and semantics is provided in Section II. In Section III, the syntax and semantics of a generalized time-triggered model are given, and a simulation theorem is stated. In Section IV, we demonstrate how to prove the schedule for a protocol realization satisfies the scheduling constraints of the time-triggered model using as a case-study the SPIDER Distributed Diagnosis Protocol (proofs for the SPIDER Clock Synchronization Protocol and schedule optimizations are provided on-line<sup>1</sup>). Concluding remarks are given in Section V.

## II. THE SYNCHRONOUS MODEL

The synchronous model presented is a variant of Lynch’s formulation [13] subsequently adapted by Rushby for the purposes of formulating it in a mechanical theorem-prover [1]. Here, we make some slight modifications to the language and also introduce a round-independence relation, to be described shortly. In the synchronous model, distributed protocols are specified as if the nodes in the distributed system execute in lock-step. A synchronous protocol proceeds in rounds. In a round, nodes synchronously and instantaneously update their outbound channels (in the *communication phase*) and then their local state (in the *computation phase*), based on the incoming messages received on their inbound channels [13].

### A. Syntax

We begin by fixing a set of messages,  $mess$ . A distinguished element  $null$  represents the absence of a message (it can also represent a “do not care” message). Let  $P$  be a nonempty set of node identifiers. For each  $p \in P$ , the following sets and total functions are defined:

- A set of node identifiers,  $out\_nbrs_p$ , identifying the *outbound neighbors*; i.e., the nodes to which  $p$  is connected by outbound channels. A set of node identifiers,  $in\_nbrs_p$ , identifying the *inbound neighbors*, can be defined from the outbound neighbors:

$$in\_nbrs_p \stackrel{df}{=} \{q \in P \mid p \in out\_nbrs_q\}$$

- A set of states,  $states_p$ . A distinguished component of the state,  $r$ , keeps track of the current round. The state  $init\_s_p$  is the initial state.
- A *message-generation function*  $msg_p : states_p \times out\_nbrs_p \rightarrow mess$  that returns the message  $p$  sends to each node to which it is connected by an outbound channel;  $null$  is returned if no message is sent.
- A higher-order *state-transition function*  $trans_p : states_p \times (in\_nbrs_p \rightarrow mess) \rightarrow states_p$  that returns the new state of  $p$  based on the current state and inputs generated by its inbound neighbors.

Sometimes we omit the node-identifier subscript from a set or function to denote a global representation of the system. For example, we define the *global state* to be the function  $states \stackrel{df}{=} \lambda p. states_p$ .

Finally, we introduce a *round-independence* relation *independent* over rounds that holds if messages to be sent in  $r + 1$  do not depend on the computation that occurs during round  $r$ . This relation is used to determine whether a messages for the subsequent round can be sent before computation in the current round is complete. We call this *round-based pipelining*.

### B. Semantics

The semantics of a synchronous specification can be given by a transition system expressed as a recursive function. The communication phase is modeled by each node applying its  $msg$  function, and the computation phase is modeled by each node applying its  $trans$  function. The function  $run$  takes the number of rounds of execution and the global initial state and returns the final global state ( $run_p$  is  $p$ ’s component of the global state returned by  $run$ ). Thus, for the initial round  $init\_rnd$  and the initial state  $init\_s$  of a protocol, its behavior can be defined as  $run(init\_rnd, init\_s)$ , where

$$run(r, s) \stackrel{df}{=} \begin{cases} s & \text{if } r = 0 \\ \lambda p. trans_p(run_p(r-1, s), \\ \lambda q. msg_q(run_q(r-1, s), p)), & \text{else} \end{cases}$$

where  $q \in in\_nbrs_p$

The protocols we model execute for only a finite number of rounds. However, a protocol may be scheduled to execute an infinite number of times.

The meaning of the round-independence relation is captured by Axiom 1, which describes the behavior of pipelined communication and computation phases by stating that if the relation holds at round  $r$ , then the messages generated from the states in rounds  $r$  and  $r - 1$  are equivalent. The intuition is that if the computation phase of  $r$  does not depend on the messages

sent in the communication phase of  $r$ , then the computation phase may begin before the computation phase ends. We motivate the use of the relation in pipelining optimizations in Section III-0e.

*Semantic Axiom 1 (Pipelining):*

$$\begin{aligned} & \neg \text{independent}(0) \\ \text{and } & \left( \begin{array}{l} \text{independent}(r) \\ \text{implies } (\forall q \in \text{out\_nbrs}_p : \\ \quad \text{msg}_p(\text{run}_p(r, \text{init}_s), q) \\ \quad = \text{msg}_p(\text{run}_p(r-1, \text{init}_s), q)) \end{array} \right) \end{aligned}$$

### III. THE GENERALIZED TIME-TRIGGERED MODEL

We extend the synchronous model presented in the previous section to take into account the real-time behavior of the protocols' execution. Some of the syntax comes directly from the original model Rushby developed [1]; the syntactic extensions we introduce for the time-triggered model are noted specifically. After the extensions, we describe the semantics of the model and then present a simulation theorem between an arbitrary protocol in the synchronous and time-triggered models.

Here, we take a moment to motivate informally the generalizations to Rushby's original time-triggered model. We use these generalizations to model the NASA SPIDER protocols and their realizations. We do not know to what extent these generalizations support current realizations of similar time-triggered systems, such as SAFEbus, TTA, and FlexRay [4], but the generalizations can be used to explore more aggressive timing characteristics for any time-triggered system satisfying the model.

Recall the generalizations for which we make provisions: (1) event-triggered behavior, (2) communication delays, (3) reception windows, (4) non-static clock skew, and (5) pipelined rounds; we motivate them in the same order.

*a) Event-Triggered Behavior:* Some protocols, while mostly time-triggered, occasionally manifest *event-triggered* behavior—actions driven by the observance of some event rather than reaching some pre-scheduled clock-time. A typical example is a clock synchronization protocol such as Davies and Wakerly's protocol [14] or Srikanth and Toueg's protocol [15]. Some of the messages sent in the protocols may be determined by the global schedule, but others are event-triggered: when a node receives some number of messages over its inbound channels, it sends a synchronization (or *echo*) message.

*b) Communication Delays:* Communication is not instantaneous; latency depends on both the distance a message travels and the medium through which it travels. Latency must be accounted for in tightly synchronized systems with large differences in latency between nodes. Furthermore, for a given distance and medium, there is a nominal latency, and error bounds are also introduced to bound the greatest deviation from the nominal latency that is not regarded as a faulty communication.

*c) Reception Windows:* Based on the anticipated send time, the expected latency, and the local clock-time, a receiving node will open a *reception window*, which is the set of clock ticks during which the node allows incoming messages in a given round. Messages received outside the window are marked as being faulty. We introduce reception windows into the model to ensure that the windows in a realization do not violate the synchrony assumption.

*d) Non-Static Clock Skew:* Provisions for reasoning about non-static clock skew have two benefits. First, they allow protocols that satisfy the assumptions of the time-triggered model but nevertheless directly affect the system's timing characteristics (e.g. clock synchronization, self-stabilization, and startup protocols [11]) to be specified in a time-triggered model rather than a more general asynchronous model. Second, they allow for formal reasoning about schedule optimizations. Time-triggered system schedules (also known as *task-descriptor lists* [5]) are usually designed with respect to the maximum possible clock skew during the normal operation of the system. When clocks are not resynchronized, the maximum possible clock skew increases as a linear function of time. If the difference between the possible clock skew at different points in the system's execution is significant, then a schedule can be tightened at those points that the clock skew is small.

*e) Pipelining Rounds:* Embedded control systems often have hard real-time deadlines that may require aggressive schedules. It may be possible to pipeline the communication and computation rounds of a single protocol or of multiple protocols for better throughput; we call pipelining of this sort *round-based pipelining*. For rounds of the schedule satisfying Axiom 1, the computation phase can begin before the communication phase has completed. Section IV-6 mentions how to exploit this in schedules that interleave distinct protocols.

#### A. Syntax

We define *real-time* to be the set of real numbers  $\mathbb{R}$  and *clock-time* to be the set of integers  $\mathbb{Z}$ . Real-time is measured in some arbitrary unit of time (e.g. milliseconds), and clock-time is measured in ticks. By convention, real-time variables and constants are lower-case and clock-time variables and constants are upper-case.

A time-triggered specification extends the syntax for a synchronous specification as follows. (The syntax deals with both the protocol specification and its time-triggered implementation.)

Let  $P$  be a nonempty set of node identifiers. For each  $p \in P$ , the following total functions are defined:

- An *inverse clock* function  $C_p : \mathbb{R} \rightarrow \mathbb{Z}$  that takes a real-time as an input and returns a clock-time.
- A *schedule* function  $\text{sched}_p : \mathbb{N} \rightarrow \mathbb{Z}$  from rounds to clock-times. It parameterizes the communication and computation phase schedules, defined as offsets from the beginning of the round. To accommodate event-triggered behavior, we take a more general view of the schedule function than Rushby does: the schedule function may *determine* the time at which some event occurs, for a

time-triggered action, or it may simply denote the clock-time at which an event occurs, for an event-triggered action.

- A relation  $sent_p \subseteq out\_nbrs_p \times mess \times \mathbb{R}$ , the tuples of which consist of a node  $q$  (that is an outbound neighbor of  $p$ ), a message  $m$ , and a real-time  $t$  and holds if  $p$  sent message  $m$  to  $q$  at real-time  $t$ .
- A relation  $recv_p \subseteq in\_nbrs_p \times mess \times \mathbb{R}$ , the tuples of which consist of a node  $q$  (that is an inbound neighbor of  $p$ ), a message  $m$ , and a real-time  $t$  and holds if  $p$  received message  $m$  from  $q$  at real-time  $t$ .

In addition, the following functions and constants, not parameterized by node identifiers, are also defined:

- A *schedule discrepancy* function  $\Lambda : \mathbb{N} \rightarrow \mathbb{Z}$  from rounds to clock-times denoting the maximum clock-time discrepancy between the schedule functions for that round. This function is added to Rushby’s model since nodes may not share the same schedule (due to event-triggered behavior).
- A *communication offset* function  $D : \mathbb{N} \rightarrow \mathbb{Z}$  from rounds to a clock-time offset. It determines the clock-time at which nodes send messages in each round.
- A *communication delay*  $\delta_{nom} > 0$  is a real-time constant that denotes the expected nominal delay between when a message is sent and when it is received. The small real-time constants  $e_l > 0$  and  $e_u > 0$  denote the maximum offsets from  $\delta_{nom}$  at which a message is received sooner ( $\delta_{nom} - e_l$ ) or later ( $\delta_{nom} + e_u$ ) than expected, respectively. We require  $e_l < \delta_{nom}$  and  $e_u < \delta_{nom}$ . These constants, added to Rushby’s model, provide finer-grained reasoning on the real-time bounds of latency-sensitive time-triggered protocols (e.g., clock synchronization protocols).
- A *computation offset* function  $P : \mathbb{N} \rightarrow \mathbb{Z}$  from rounds to a clock-time offset. It determines the clock-time at which nodes begin computation in each round.
- A *maximum drift rate*  $\rho \in \mathbb{R}$  such that  $0 < \rho < 1$ . This is the maximum rate at which a clock may drift.
- A dynamic *clock skew*  $\Sigma(r) \geq 0$  function is introduced to Rushby’s model, which denotes the greatest clock-time skew occurring between a sender and receiver during the duration of round  $r$ .
- A *reception window* function  $R : \mathbb{N} \rightarrow \mathbb{Z}$  from rounds to a *reception window offset* is also introduced to Rushby’s model. It marks the clock-time at which a node accepts inbound messages. In round  $r$ , the reception window closes at  $P(r)$ .

### B. System Assumptions and Schedule Constraints

We constrain the interpretations that can be given to the syntax when defining a time-triggered system with the following *system assumptions* and *schedule constraints*. The system assumptions describe the assumed behavior of the underlying system—most notably, the behavior of the local clocks. The schedule constraints ensure the schedule of time-

triggered events, given the system assumptions, gives rise to synchronous behavior.

1) *System Assumptions*: As in Rushby’s original model, we present four system assumptions [1]; recalling that three of Rushby’s formulations were inconsistent [10], we present mended and generalized assumptions here. As usual, free variables are implicitly universally-quantified.

Assumption 1 bounds the maximum drift of a clock in terms of the maximum drift rate,  $\rho$ .

*System Assumption 1 (Clock Drift Rate)*: Let  $t_1 \geq t_2$ . Then  $\lfloor (1 - \rho) \cdot (t_1 - t_2) \rfloor \leq C_p(t_1) - C_p(t_2) \leq \lceil (1 + \rho) \cdot (t_1 - t_2) \rceil$ .

Lemma 1 shows that the clocks are monotonic.

*Lemma 1*:  $t_1 < t_2$  implies  $C_p(t_1) \leq C_p(t_2)$ .

*Proof*: By System Assumption 1,  $C_p(t_2) \geq C_p(t_1) + \lfloor (1 - \rho)(t_2 - t_1) \rfloor$ . ■

Assumption 2 ensures the skew between clocks is no greater than the maximum clock skew so that if any clock is in the communication phase of round  $r$ , then all of the clocks are synchronized within the skew of that round. Clock-time  $sched_p(r) + D(r)$  is the clock-time at which  $p$  sends its messages in round  $r$ , and  $sched_p(r) + P(r)$  is the clock-time at which it begins the computation phase of round  $r$ .

*System Assumption 2 (Clock Synchronization)*:

$$\begin{aligned} & \left( \begin{aligned} & \max(C_p(t), C_q(t)) \\ & \geq \min(sched_p(r), sched_q(r)) + D(r) \\ & \text{and } \min(C_p(t), C_q(t)) \\ & \leq \max(sched_p(r), sched_q(r)) + P(r) \end{aligned} \right) \\ & \text{implies } |C_q(t) - C_p(t)| \leq \Sigma(r) \end{aligned}$$

Assumption 3 ensures that messages are received within the communication delay of when they are sent, modulo error, and that messages received were not “spontaneously generated.”

*System Assumption 3 (Maximum Communication Delay)*:

There exists some real-time  $d$ , where  $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$ , such that  $sent_p(q, m, t)$  if and only if  $recv_q(p, m, t + d)$ , and there exists some real-time  $d'$ , where  $\delta_{nom} - e_l \leq d' \leq \delta_{nom} + e_u$ , such that  $recv_q(p, m, t)$  if and only if  $sent_p(q, m, t - d')$ .

Assumption 4 constrains the maximum discrepancy permitted between the schedule functions of two nodes for a given round. For a particular implementation, whether this constraint is met depends on the constraints for the event-triggered behavior of the individual nodes.

*System Assumption 4*:  $0 \leq |sched_p(r) - sched_q(r)| \leq \Lambda(r)$ .

2) *Schedule Constraints*: We present six schedule constraints to ensure the time-triggered schedule implements a synchronous system. Constraints 1 - 3 generalize Rushby’s original constraints [1], and constraints 4 - 6 are new constraints necessary to constrain pipelining and the scheduling of receivers’ reception windows. The schedule constraints are what we later prove hold of the time-triggered schedules in Section IV.

Constraint 1 ensures that the computation offset of round  $r$  falls within round  $r$ .

*Schedule Constraint 1 (Offset Constraint):*  $0 < P(r) < sched(r+1) - sched(r)$ .

Schedule constraint 2 gives the minimum communication offset. Note that if the nominal delay is substantially larger than the clock skew (as is the case in tightly-synchronized systems), the skew has little bearing on when messages can be sent.

*Schedule Constraint 2 (Communication Constraint):*  
 $D(r) \geq \Sigma(r) + \Lambda(r) - \lfloor (1-\rho) \cdot (\delta_{nom} - e_l) \rfloor$ .

Similarly, constraint 3 gives the minimum computation offset. The offset must be greater than the latest time at which a non-faulty message may arrive, which is the sum of the communication offset, the clock skew, the maximum schedule discrepancy, and the maximum delay.

*Schedule Constraint 3 (Computation Offset Constraint):*  
 $P(r) > D(r) + \Sigma(r) + \Lambda(r) + \lceil (1+\rho) \cdot (\delta_{nom} + e_u) \rceil$ .

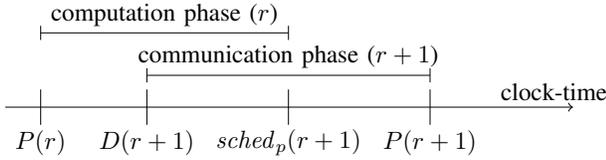


Fig. 2. Pipelined Communication Phase (Constraint 5)

Constraint 4 ensures that pipelining only occurs when the messages to be sent do not depend on the computations from the previous round, and constraint 5 restricts pipelining to consecutive rounds. The effect of pipelining is illustrated in Figure 2.

*Schedule Constraint 4:*  $\neg independent(r)$  implies  $D(r) \geq 0$ .

*Schedule Constraint 5:*  $r > 0$  implies  $D(r) \geq P(r-1) - sched(r) + sched(r-1)$ .

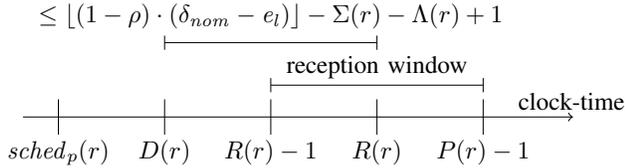


Fig. 3. Reception Window (Schedule Constraint 6)

The final schedule constraint, Constraint 6, restricts when the reception window is opened. The constraint is illustrated in Figure 3. The reception window must be opened soon enough so that non-faulty messages are received within the window. The formula  $\lfloor (1-\rho) \cdot (\delta_{nom} - e_l) \rfloor$  gives a lower bound on the minimum message delay. We add  $D(r)$  to take into account the clock-time offset at which the message is sent. The skew for the round,  $\Sigma(r)$ , is subtracted to account for the case where the receiver's clock is maximally faster than the sender's. A one-tick constant is added to the upper bound on  $R(r)$  because

the reception window is opened on a clock edge, but messages arrive asynchronously. A message that arrives strictly less than one clock tick before the reception window is opened will be latched on the clock edge when the window is opened.

*Schedule Constraint 6 (Reception Window Constraint):*  
 $0 \leq R(r) \leq D(r) + \lfloor (1-\rho) \cdot (\delta_{nom} - e_l) \rfloor - \Sigma(r) - \Lambda(r) + 1$ .

### C. Semantics

The semantics for the time-triggered model is a transition system in which states are pairs of the form  $\langle s, t \rangle$ , where  $s$  is a global state of the system together with the current real-time,  $t$ . The transitions between states are constrained by the axioms given in this section as well as Axiom 1, from the synchronous model.

The axioms are defined over the following uninterpreted functions. We give the type signatures of the functions, as well as their intended interpretations:

- A function  $sendtime_p : \mathbb{N} \rightarrow \mathbb{R}$  from rounds to real-times denoting the real-time that  $p$  broadcasts messages in each round.
- A *time-triggered system state* function  $ttss_p : states \times \mathbb{Z} \rightarrow states_p$  that takes a global state  $s$ , a clock-time  $T$ , and returns  $p$ 's state after executing for  $T$  clock ticks from  $s$ .
- A *time-triggered inbound messages* function  $ttin_p : \mathbb{Z} \times in\_nbrs_p \rightarrow mess$  that maps a clock-time  $T$  and an inbound neighbor  $q$  to the message  $p$  receives from  $q$  at  $T$ .
- A *time-slice* function  $gs : \mathbb{N} \rightarrow \mathbb{R}$  from rounds to real-times. Its purpose is to provide real-times at which the system state of the time-triggered model of a protocol is the same as the untimed model of the protocol, for each round.

Axiom 2 constrains the  $sendtime_p$  function by ensuring that at the real-time that  $p$  broadcasts its message in round  $r$ , its clock-time is at the communication offset into that round.

*Semantic Axiom 2:*  $C_p(sendtime_p(r)) = sched_p(r) + D(r)$ .

Axioms 3 and 4 constrain the behavior of the  $sent_p$  function by first stating the sufficient conditions for it to hold and then the necessary conditions for it to hold. Axiom 3 ensures that the message  $p$  sends to  $q$  at the real-time  $sendtime_p(r)$  is the message generated by its message-generation function using its time-triggered state at the beginning of round  $r$ . Axiom 4 ensures that if the  $sendtime_p$  relation is satisfied, then it is satisfied by a message generated by the message-generation function in some round and by the real-time at the communication delay into the round.

*Semantic Axiom 3:*  $sent_p(q, msg_p(ttss_p(s, sched_p(r) + D(r)), q), sendtime_p(r))$ , where  $q \in out\_nbrs_p$ .

*Semantic Axiom 4:*  $sent_p(q, m, t)$  implies there exists a round  $r$  such that  $t = sendtime_p(r)$  and  $m = msg_p(ttss_p(s, sched_p(r) + D(r)), q)$ , where  $q \in out\_nbrs_p$ .

Before stating the next axiom, we define the relation  $recv\_win\_open_p$ , which takes a real-time  $t$  and a round  $r$  and is true if the real-time falls within  $p$ 's reception window for



```

REALTIME : TYPE = REAL;
CLOCKTIME : TYPE = INTEGER;
OFFSET : TYPE = {T: CLOCKTIME | T >= 0};
RND : TYPE = NATURAL;
rho : REALTIME = 1/10000;
d_nom : {t: REALTIME | t >= 0} = 5;
ERROR : TYPE = {t: REALTIME |
  t >= 0 AND t < d_nom};
e_l : ERROR = 5/10000;
e_u : ERROR = 5/10000;
% floor((1 - rho) * (d_nom - e_l))
fl_d_min : CLOCKTIME = 4;
% ceiling((1 + rho) * (d_nom + e_u))
cd_d_max : CLOCKTIME = 6;

```

Fig. 4. Type and Constant Declarations

2) *Variables*: In SAL, we build a state-machine to model-check. The state-machine transitions follow the order of the schedule’s rounds and update state variables accordingly. Therefore, in the SAL model, we replace some of the mathematical functions from the time-triggered system model presented in Section III with corresponding state variables ranging over rounds. Thus, the set of state variables include  $sched$ ,  $D$ ,  $P$ ,  $R$ ,  $\Lambda$ ,  $\Sigma$ , *independent*, and  $R$ . The state variables may be nondeterministically updated in the state-machine transitions depending on the specifics of the schedule being verified. In the schedule verified for the SPIDER DD Protocol, the values of  $D$ ,  $P$ ,  $R$ , and  $\Sigma$  are constant over the rounds for this protocol’s schedule; only  $sched$  is updated from round to round. The other protocol schedule verifications are more complex; see IV-6.

3) *Schedule Constraint Specification*: The schedule constraints stated in Section III-B2 are stated in SAL as shown in Figure 5. Some of these constraints compare the schedule between successive rounds (e.g., Constraint 1). Because we have transcribed the functions over the rounds to variables that are updated in the state machine at each round, these relations may take as arguments the values of these variables in a round and compare them to the values in the next round (e.g., `constraint1` takes `pre_sched` and `sched` as arguments, denoting the values for  $sched(r-1)$  and  $sched(r)$ , respectively). The SPIDER DD Protocol contains no event-triggered behaviors; therefore, for all rounds, the schedule skew  $\Lambda$  is zero. We therefore omit it from the constraints.

4) *Specifying a Round-Based Schedule*: We create a state-machine representation of how the schedule constraints evolve through the rounds of execution. In addition to schedule variables, for each constraint, a boolean variable is declared. The value of the variable is determined by whether its associated schedule constraint is satisfied in the present round. The state machine includes a counter  $r$  that records the current round in the synchronous abstraction of the protocol. In each initial state, this counter is set to 0. Each transition from a state in round  $r$  is to a state in round  $r+1$ . In general, we check the schedule constraints for the next-state values of the variables. For constraints that compare the values between rounds, the current-state variable values and the next-state variable values

```

constraint1(P: OFFSET, pre_sched: CLOCKTIME,
  sched: CLOCKTIME): BOOLEAN =
  0 < P AND P < sched - pre_sched;

constraint2(D: CLOCKTIME, S: OFFSET): BOOLEAN =
  D >= S - fl_d_min;

constraint3(P: OFFSET, D: CLOCKTIME, S: OFFSET):
  BOOLEAN = P > D + S + cd_d_max;

constraint4(r: RND, D: CLOCKTIME): BOOLEAN =
  (NOT independent?(r)) => D >= 0;

constraint5(pre_P: OFFSET, D: CLOCKTIME,
  pre_sched: CLOCKTIME, sched: CLOCKTIME):
  BOOLEAN = D >= pre_P - sched + pre_sched;

constraint6(D: CLOCKTIME, R: CLOCKTIME, S: OFFSET):
  BOOLEAN = R - 1 <= D + fl_d_min - S;

```

Fig. 5. SAL Specification of the Generalized System Assumptions

are compared in the constraint. Because there are no previous state assignments in round 0, those state variables associated with constraints that compare values between rounds are declared to be true upon initialization.

Not every state variable needs to be updated in each transition. If a state variable is not reassigned in a guarded transition, its value remains the same in the next state.

5) *Verification*: The property stating that in all reachable states, each constraint holds can then be specified by the following LTL state invariant.

```

constraints: LEMMA SYSTEM |-
  G(c1 AND c2 AND c3 AND c4 AND c5 AND c6);

```

The property is verified by executing SAL’s infinite-state bounded model checker. The lemma `constraints` is verified by the  $k$ -induction solver, for  $k = 2$ . The proof is fully automatic and requires no supporting invariants.

6) *Other Verifications*: We have also used this proof technique to verify the SPIDER Clock Synchronization Protocol schedule. The purpose of the clock synchronization protocol is to resynchronize the local clocks, in the presence of faults, after they have possibly drifted apart [11]. Consequently, the schedule of the clock synchronization protocol is more complex, as the skew is a function of the round of the protocol.

Similarly, we have also verified a schedule interleaving distinct SPIDER protocols. In these schedules, we can take advantage of the round-based pipelining optimizations. Both of these verifications are available on-line.<sup>1</sup>

## V. DISCUSSION

Faults are not dealt with explicitly in the models presented; we discuss them below. Concluding remarks follow.

### A. Faults

Neither the synchronous nor time-triggered model presented explicitly model faulty behavior. Nevertheless, because many

time-triggered protocols are fault-tolerant, we ultimately wish to prove that the protocols behave correctly in the presence of faults. Before discussing faults specifically, recall that the state-transition function *trans* and the message-generation function *msg* are left uninterpreted, and the same functions appear in both the synchronous and time-triggered models. Thus, the simulation theorem holds regardless of their instantiations; in particular, the functions can be partially-interpreted and under-specify a protocol, and the theorem still holds.

Keeping this in mind, faults can be modeled, as Rushby notes, by partially-interpreting *trans* and *msg*, allowing them to return arbitrary values, nondeterministically, if a node or channel is faulty [1]. In fact, all faulty behavior can be modeled, with no loss of fidelity, by partially-interpreting the message-generation function only [18].

In the synchronous model, the correctness of a protocol can be verified under a *maximum fault assumption* (MFA), which constrains the kinds of faults, and the number of each kind, under which the protocol is hypothesized to behave correctly [1]. If the effects of faults are captured by the message-generation function *msg*, then the MFA can be thought of as a constraint on the function's nondeterminism. Thus, one purpose of the time-triggered model is to *define* the timing behaviors that are non-faulty. That is, if the timing characteristics of the system satisfy the system assumptions and schedule constraints of the time-triggered model, then no faults will result if these characteristics hold in a realization (timing and value faults may still arise from environment factors).

## B. Concluding Remarks

The approach presented herein and illustrated in Figure 1 is one portion of an end-to-end verification methodology—from the distributed protocols to the hardware implementations of the nodes—in a time-triggered system. As an anonymous reviewer notes, the results presented herein can be combined with recent work in physical-layer protocol verification [19] and gate-level I/O device verification [20] to further the goal of an end-to-end verification of time-triggered systems.

## VI. ACKNOWLEDGMENTS

This paper is a revised portion of a recent dissertation [16]. We thank Steve Johnson (who advised the dissertation), Paul Miner, Geoffrey Brown, Larry Moss, Wilfredo Torres-Pomales, and our anonymous reviewers for their ideas and comments. Finally, this paper owes a large debt to John Rushby's original work on the topic.

## REFERENCES

- [1] J. Rushby, "Systematic formal verification for fault-tolerant time-triggered algorithms," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 651–660, September 1999.
- [2] P. Koopman, Ed., *Critical Embedded Automotive Networks*, ser. IEEE Micro, vol. 22–4. IEEE Computer Society, July/August 2002.
- [3] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire - a total approach to dependability," in *IFIP Congress Topical Sessions*, 2004, pp. 191–212.
- [4] J. Rushby, "Bus architectures for safety-critical embedded systems," in *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, ser. Lecture Notes in Computer Science, T. Henzinger and C. Kirsch, Eds., vol. 2211. Lake Tahoe, CA: Springer-Verlag, Oct. 2001, pp. 306–323.
- [5] H. Kopetz, *Real-Time Systems*. Kluwer Academic Publishers, 1997.
- [6] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993, available at <http://citeseer.nj.nec.com/butler93infeasibility.html>.
- [7] P. Miner, A. Geser, L. Pike, and J. Maddalon, "A unified fault-tolerance protocol," in *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, ser. LNCS, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Springer, 2004, pp. 167–182, available at [http://www.cs.indiana.edu/~lepik/pub\\_pages/unified.html](http://www.cs.indiana.edu/~lepik/pub_pages/unified.html).
- [8] H. Pfeifer, "Formal analysis of fault-tolerant algorithms in the time-triggered architecture," Ph.D. dissertation, Universität Ulm, 2003, available at <http://www.informatik.uni-ulm.de/ki/Papers/pfeifer-phd.html>.
- [9] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, February 1995.
- [10] L. Pike, "A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms"," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 347–348, May 2006, available at [http://www.cs.indiana.edu/~lepik/pub\\_pages/time\\_triggered.html](http://www.cs.indiana.edu/~lepik/pub_pages/time_triggered.html).
- [11] W. Torres-Pomales, M. R. Malekpour, and P. Miner, "ROBUS-2: A fault-tolerant broadcast communication system," NASA Langley Research Center, Tech. Rep. NASA/TM-2005-213540, 2005.
- [12] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer-Aided Verification, CAV 2004*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Boston, MA: Springer-Verlag, July 2004, pp. 496–500.
- [13] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [14] D. Davies and J. F. Wakerly, "Synchronization and matching in redundant systems," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 531–539, June 1978.
- [15] T. K. Srikant and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, July 1987.
- [16] L. Pike, "Formal verification of time-triggered systems," Ph.D. dissertation, Indiana University, 2006, available at <http://www.cs.indiana.edu/~lepik/phd.html>.
- [17] L. Pike and S. D. Johnson, "The formal verification of a reintegration protocol," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM Press, 2005, pp. 286–289, available at [http://www.cs.indiana.edu/~lepik/pub\\_pages/emsoft.html](http://www.cs.indiana.edu/~lepik/pub_pages/emsoft.html).
- [18] L. Pike, J. Maddalon, P. Miner, and A. Geser, "Abstractions for fault-tolerant distributed system verification," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., vol. 3223. Springer, 2004, pp. 257–270, available at [http://www.cs.indiana.edu/~lepik/pub\\_pages/abstractions.html](http://www.cs.indiana.edu/~lepik/pub_pages/abstractions.html).
- [19] G. M. Brown and L. Pike, "Easy parameterized verification of biphasic mark and 8N1 protocols," in *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, 2006, pp. 58–72, available at [http://www.cs.indiana.edu/~lepik/pub\\_pages/bmp.html](http://www.cs.indiana.edu/~lepik/pub_pages/bmp.html).
- [20] S. Knapp and W. Paul, "Realistic worst case execution time analysis in the context of pervasive system verification," in *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, ser. LNCS volume 4444. Springer, 2006, pp. 53–81.

# A Mechanized Refinement Framework for Analysis of Custom Memories

Sandip Ray  
University of Texas at Austin  
sandip@cs.utexas.edu

Jayanta Bhadra  
Freescale Semiconductor Inc.  
jayanta.bhadra@freescale.com

**Abstract**— We present a framework for formal verification of embedded custom memories. Memory verification is complicated by the difficulty in abstracting design parameters induced by the inherently analog nature of transistor-level designs. We develop behavioral formal models that specify a memory as a system of interacting state machines, and relate such models with an abstract read/write view of the memory via refinements. The operating constraints on the individual state machines can be validated by readily available data from analog simulations. The framework handles both static RAM (SRAM) and flash memories, and we show initial results demonstrating its applicability.

## I. INTRODUCTION

This paper describes an approach to verify embedded custom memories. Memory verification entails showing that a transistor implementation conforms to the high-level view of a state machine that stores and retrieves data at addressed locations. Memories are complex analog artifacts, optimized for performance, area, power, etc., and account for about half the real estate and more than 50% of the transistor count of a microprocessor. This makes their verification a critical component of the overall design validation. However, given the size and complexity of a custom memory core, it is impossible to validate the entire core by analog simulation. Thus, a key challenge is to derive an effective abstraction which can be formally compared against the high-level specification.

The common approach to abstract a traditional SRAM is to extract a *switch-level* model [1] that represents the memory netlist as a set of *nodes* connected by transistor *switches*. Each node has state 0, 1, or X; each switch has state “open”, “closed”, or “indeterminate”; state transitions are specified by *switch equations*. These models capture many aspects of transistor circuits, namely bidirectionality, signal strengths, etc. The common analyzers for constructing such models are the ANAMOS [2] and its variants; they partition a netlist into *channel connected subcomponents* (CCSs) and analyze each component separately to construct the switch equations.

However, in spite of their sophistication, switch-level analyzers ignore many analog effects. For instance, the *strength assignment* procedure in ANAMOS produces a significant mismatch with detailed analog simulations for netlists containing transistors of closely matching but different strengths [3]. While these discrepancies can be ameliorated by designing more and more accurate analyzers [4], such an approach does not solve the fundamental problem of effectively representing inherently analog behaviors with equations in a discrete

algebra. The problem is exacerbated with the advent of flash memories that contain both regular and *Floating Gate* (FG) transistors; FG transistors “break” the view of netlists as a collection of switches, making switch-level analysis untenable.

In this paper, we present a new approach to abstracting memory implementations. Instead of extracting a switch-level model by *structural analysis* of a transistor netlist implementing a memory core, we formalize its *behavior* as a system of interacting state machines. The viability of the method is based on the observation that in an industrial design flow, custom memories are designed *not* as an ad-hoc transistor network but by interconnecting small, cohesive units such as *bitcells*, *sense amplifiers*, etc., with well-understood electrical properties in their range of operation. For instance, in Motorola’s design flow, the units are carefully architected to operate over a limited sequence of *certified* stimulus patterns, each of which is validated by extensive analog SPICE simulation across various process corners and operating conditions [3]. Thus, it is natural to formalize the *behavior* of each unit as a state machine using guarded transitions that encode its operating constraints. This enables us to reduce a memory implementation to a formal behavioral model specifying an interacting composition of state machine components, with each component representing the behavioral model for a pre-computed unit. Note that by focusing on the *behavior* of the units, our approach is agnostic to the nature of the transistors (standard, FG, etc.) used in the implementation, making it applicable to both SRAM and flash designs. Finally, we show how to prove a refinement theorem relating such compositions with high-level memory specifications using an assume-guarantee technique.

Our approach is mechanized in the ACL2 theorem prover [5]. We show how the use of the expressive language of a general-purpose theorem prover enables effective compositional reasoning about the interacting state machines.

## II. MODELING CUSTOM SRAM

We illustrate our approach with the bitcell implementation shown in Fig. 1. From an *electrical perspective*, reading from the bitcell can be explained by the following operations.

- Initially, the precharge (*pch*) signal turns 1 in order to precharge the bitline (*bl*) and bitline-bar (*blb*) to 1s.
- Next, the wordline (*wl*) turns 1 indicating that the decoded address matches that of the bitcell. Thus the data stored in the bitcell (say 0) and its complement (1) are gated to

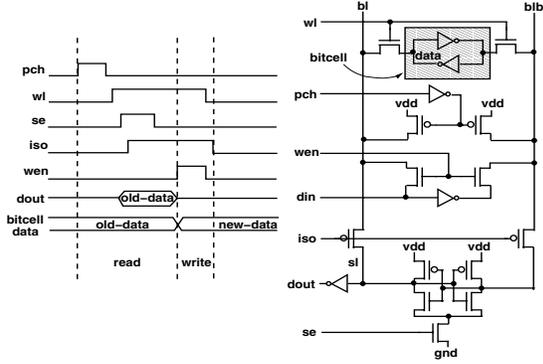


Fig. 1. A Standard SRAM Bitcell and Associated Timing Diagram

$bl$  and  $blb$  respectively. Due to the large load on  $bl$ , the bitcell cannot pull it down from 1 to 0;  $bl$  is pulled down to an indeterminate value  $(1 - \delta)$  (where  $0 < \delta < 1$ ), while  $blb$  retains a 1. Furthermore, since the isolate ( $iso$ ) signal is 0, the sense-line ( $sl$ ) is pulled down to  $(1 - \delta)$ .

- Finally, the sense ( $se$ ) signal, followed by  $iso$ , turn 1 resulting in (i) an electrical disconnection between  $bl$  and  $sl$ , and (ii) converting the bottom 5-transistor pack into a latch that pulls the value of  $sl$  from  $(1 - \delta)$  to a solid 0. The value is then inverted and is obtained at  $dout$ .

Note that reading a bitcell is a complex analog problem — how to drive a *large* load (a bitline) with a bitcell that is quite *small* in strength. Furthermore, the circuit functionality depends on the constraints on the relative times within which successive signals change value. For example,  $wl$  must be 1 for a pre-determined bounded interval in order for the bitcell value to drive  $bl$  from 1 to  $(1 - \delta)$ . Nevertheless, the *behavior* of the signal pattern can be viewed as discrete state transitions: if the data stored in the bitcell is 0, then under appropriate conditions  $bl$  transits from state 1, through  $(1 - \delta)$  and so on until finally the circuit produces 1 at  $dout$ .

Our approach works on the bitcell design as follows. We identify the design as a composition of state machines that correspond to the following two components:

- 1) the *bitcell* (from the shaded region together with the structure of the  $pch$  and  $iso$ ), with the  $wl$ ,  $pch$ , and  $iso$  signals as input and  $bl$  and  $blb$  as output, and
- 2) the *sense amplifier* (from the 5-transistor pack together with the transistors gated by  $iso$ ), with  $bl$  and  $blb$  as inputs and  $dout$  as the output.

Each state machine is pre-computed into a “library”. To make the library *generic*, the state machines are parameterized to work over a range of operating constraints. For instance, to model the time delay between  $pch$  and  $iso$  signals, the bitcell component contains parameters  $n_0$ ,  $n_1$ , and  $n_2$  (among others), with constraints that on a *read*, (i)  $pch$  is 1 and  $iso$  is 0 for at least  $n_0$  units, (ii) both  $pch$  and  $iso$  are 0 for at least  $n_1$  units thereafter and  $wl$  becomes 1 in this interval, and (iii)  $iso$  is 1 for at least  $n_2$  units subsequently. For the reader familiar with ACL2, the parameters are modeled using encapsulation

(with constraints specifying operating conditions) and can be functionally instantiated [6] for concrete models.

We have used our library to model the transistor implementation of a simple but complete SRAM core consisting of an array of memory words, each word composed of a row of the bitcells in Fig. 1 (the sense amplifier being shared along a *bitcell column*), together with a decoder implementation. This example demonstrates scalability of the approach. A strength of the approach is compositionality: the extracted formal model of the memory core is merely a hierarchical composition of those of the individual bitcells, with intermediate state machines specifying the behavior of the glue logic.

### III. MODELING FLASH MEMORY

Although we can abstract custom SRAMs, our principal goal is to develop a framework for handling flash memories. The additional complexity in flash arises from FG transistors, which have, in addition to the conventional *drain* (D), *gate* (G) and *source* (S) terminals, a *floating gate* (F) — a polysilicon layer inserted in the oxide between the gate and the substrate that is physically disconnected from both S and D. A detailed treatment of flash memories is provided by Cappelletti *et al* [7]. The key electrical effect is the capacitive coupling between G, F, and the substrate. The capacitance is exploited to design a bitcell with a *single* FG transistor as follows. Controlling the stored charge in the capacitive coupling allows dynamic regulation of the threshold voltage  $V_{th}$  (the minimum voltage to turn on the device); a low threshold voltage ( $V_{th}^L$ ) represents logic 1 and high threshold voltage ( $V_{th}^H$ ) represents logic 0. Additionally, some flash designs make use of *multiple*  $V_{th}$  levels to store 2 or 3 bits in one FG transistor; we do not consider multi-level flash in this paper.

Unfortunately, the capacitive coupling mentioned above breaks the simple view of a transistor as an on/off switch, as taken by ANAMOS-like analyzers, and makes it infeasible to extract precise switch-level abstractions. Consequently, current industry practice on flash validation amounts to (i) simulating the high-level model along with the encompassing SoC, and (ii) simulating individual FG bitcells through SPICE simulations. In particular, *no* formal correspondence is guaranteed between the transistor netlist and high-level specification.

Before describing our behavioral models, we discuss the electrical effects of flash operations. Below we summarize the three main operations of an FG bitcell: *read*, *program* (writing 0), and *erase* (writing 1). The operations involve both the bitcell and the surrounding control logic.

- **Read:** For the selected bitcell, one applies a voltage  $v$  ( $V_{th}^L < v < V_{th}^H$ ) at G which is driven by the selected wordline, while keeping other wordlines at ground. If the cell has logic 0, the transistor does not turn on and no current flows to the associated sense amplifier; otherwise the bitcell turns on and current is detected, reading a 1.
- **Program:** The so-called *Channel Hot-Electron Injection* procedure is performed to inject negative charge into the FG, raising its  $V_{th}$  to  $V_{th}^H$ . Then there is a verification phase to ensure that  $V_{th}$  has been appreciably raised;

this is done by “reading” the cell with a gate voltage  $v (> V_{th}^H)$ . A result of 0 for the read indicates successful programming; otherwise programming is iterated until it succeeds or a specified number of attempts have been made, signalling failure in the latter case.

- **Erase:** Erasing is performed for an entire memory sector rather than one bitcell, and is based on removal of stored charge by a procedure called *Fowler-Nordheim tunneling*. The operation involves (i) *raising* the  $V_{th}$ s of the bitcells in the sector to  $V_{th}^H$  by programming, (ii) charge removal to lower all the  $V_{th}$ s to  $V_{th}^L$ , and finally, (iii) normalization, which employs *soft programming* to increase the  $V_{th}$  of the cells that have fallen below  $V_{th}^L$ .

The description underlines the complexity of the analog operations in a flash memory, and points to the difficulty of designing switch-level analyzers. Other factors to account for in abstracting flash memories include (i) multiple voltage levels, (ii) charge injection and removal, and (iii) complex sense amplifier activity to compare various current values. However, the *behaviors* of the individual components are still tractable (albeit more complex than SRAMs). For instance, the response of the state machine for the FG bitcell component to the electron injection phase of a *program* sequence is formalized as a non-deterministic transition raising the  $V_{th}$  by a bounded constant. Our library contains behavioral state machine models for the different components of flash memory, such as bitcell, sense amplifier, etc. Note that a few of the generic components are reused from the SRAM library.

We used behavioral abstractions to formalize a standard implementation of a NOR flash core:<sup>1</sup> bitcells are arranged in a two-dimensional array with a *row decoder* and a *column decoder*; a *read* of a bitcell causes a row to be activated by the row decoder, while the column decoder causes the appropriate column to be connected to the sense amplifier resulting in the loading of the data from the addressed bitcell to the output buffer. The extracted model is a composition of the behavioral models corresponding to the bitcells, the (row and column) decoders, the sense amplifiers, and the output buffers.

#### IV. SPECIFICATION AND VERIFICATION

We relate the executions of the memory core with a high-level specification. The specifications are abstract state machines representing the core’s interface to an SoC during functional verification. The SRAM specification supports *read*, *write*, and *reset* operations; the flash specification supports *read*, *program*, and *erase*, together with *core enable* that controls operations on the entire core, and *write protect* that regulates programming bitcells in the core.

We prove that the implementation is a *simulation refinement* [8], [9] of the specification up to stuttering, with respect to a *refinement map*. A refinement map enables us to appropriately view implementation states as specification states [10], and in our case, maps the bitcell states in the memory core

<sup>1</sup>Flash memories have two common organizations: NOR and NAND. A NOR flash is used as a nonvolatile memory with fast random access. A NAND flash can be used as a disc storage. We do not consider NAND flash.

to an association list that models the core at the specification level. We require the notion of correspondence to be stutter-insensitive to account for the timing mismatch between the implementation and specification models.

We now discuss the proof obligations. Let *rep* be a refinement map. We then define predicates *inv* and *commit*, and a function *pick* such that (i) *inv* is an implementation invariant and (ii) the following formulas are provable:

1.  $\forall s, i : inv(s) \wedge \neg commit(s, i) \Rightarrow rep(impl(s, i)) = rep(s)$
2.  $\forall s, i : inv(s) \wedge commit(s, i) \Rightarrow rep(impl(s, i)) = spec(rep(s), pick(s, i))$

Here *impl* and *spec* are the (non-deterministic) state transition functions of the implementation and specification respectively; *commit* governs for an implementation transition if the specification transits or stutters; *pick* provides the specification stimulus in case of a matching transition. The formulas above thus state that for each transition of the implementation, the specification either has a matching transition or stutters.<sup>2</sup> These proof rules, of course, can be used to compare two systems modeled at different abstraction levels; they have been adapted from Manolios’ rules for stuttering simulations [11] with the restriction that stuttering is one-sided. The restriction is justified since one step of the specification corresponds to several steps of the implementation, but not vice versa.

Using the ACL2 theorem prover we have verified the SRAM and flash models of the preceding sections. Note that each implementation is a complex composition of a large number of state machines, which normally poses a challenge to formal verification. However, the problem is ameliorated in our case by a synergy of several factors. First, the implementation correctness is independent of the *size* of the core: we replace the core size with a symbolic constant, and use symbolic rewriting of the transition relations (an area of strength of theorem provers, in particular ACL2) rather than detailed reachability analysis. A second, subtle reason arises from the nature of the models and proof obligations. The expensive verification step involves the definition and proof of the appropriate invariant *inv*. However, this step is substantially automated by using the constraints attached to the component state machines. Since the implementation is merely an interactive, hierarchical composition, the *assumed* input constraints associated with a component *C* must be implied by the invariants (*guarantees*) associated with the state machines for their environmental components. Furthermore, in a theorem prover we can define invariants with *generic*, expressive predicates. Since ACL2 supports full first order logic, we define a predicate to express (by quantification) that each state *s* is reached by transitions in which the input sequences satisfy the associated constraints. Invariance proof for this predicate reduces to the above assume-guarantee reasoning.

<sup>2</sup>We also prove that stuttering is finite, by exhibiting a well-founded ranking function that decreases along stuttering steps; this proof is trivial since timing constraints upper-bound the completion of the state transition sequence by a natural number, namely the number of delay units in system-clock cycle.

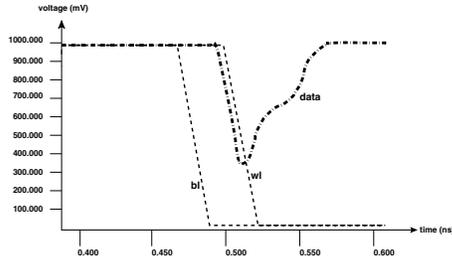


Fig. 2. A SPICE Simulation trace showing a failed write of 0 due to insufficient Setup Time

Finally, we note that one key feature of our framework is the direct behavioral correspondence between the components used for analog simulation and the formal models in our library. This facilitates corroboration of the models with readily available simulation data. Furthermore, this correspondence together with our assume-guarantee approach, can potentially identify corner cases missed in analog simulation. In one illustrative experiment, we inserted a bug in the SRAM library. During a write, the constraints on the state machines responsible for generating signals *bl* and *wl* (Fig. 1) did not guarantee that *bl* has been 0 for a sufficient time (*setup time*), before *wl* becomes 0. The bug was promptly discovered while attempting to prove the assumptions on the bitcell, and the scenario specified by the failure corresponds directly to the actual SPICE simulation pattern for the bitcell (Fig. 2).

## V. RELATED WORK AND CONCLUSION

Formalization of transistor circuits has chiefly focused on developing switch-level analyzers such as SLS [12], MOSSIM-II [1], and ANAMOS [2]. Switch-level models have found extensive applications in academia and industry [2], [13]. In addition, there has been work on equivalence verification and conservative reachability analysis of small analog circuits [14], [15], [16], [17]. Finally, the PROSYD project (<http://www.prosyd.org>) aims to provide an assertion-based run-time monitoring tool supporting STL or PSL properties in analog circuits. This tool has been applied on simulation traces from a flash memory [18].

Our framework has been inspired by recent efforts of Bhadra *et al* [3] on behaviorally formalizing transistor implementations of custom memories. They show how to abstract SRAM designs using parameterized regular expressions, and compare those abstractions with a high-level memory specification using STE. However, a limitation of that work is the difficulty to correspond the abstract models with analog simulations; our approach overcomes this by carefully constructing our library of state machine models to formalize behaviors of design units that have direct correspondence with SPICE simulations.

Our key insight is that although custom memories consist of complex analog components, the behavioral characteristics of the components are well-understood, at least within the limited range of operating conditions. By focusing on the behavior rather than the structure of components, we circumvent the

complex problem of abstracting analog operations with a discrete algebra, and formalize memory implementations as interactive compositions of relatively simple state machines. To our knowledge, ours is the first platform that permits formal analysis of *both* SRAM and flash memories. Note, however, that our approach can only be applied to memory designs constructed by interconnection of well-defined components; in particular, we *cannot* abstract an arbitrary transistor netlist.

In future work, we plan to explore if the approach scales to industrial memory designs. We also plan to extend our library of models to handle multi-level flash designs.

## Acknowledgements

Sandip Ray is funded in part by DARPA and NSF under Grant No. CNS-0429591. We thank our colleagues at Freescale Semiconductor Inc. for patiently answering our numerous questions on the electrical behavior of custom memories.

## REFERENCES

- [1] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers*, vol. C-33, no. 2, pp. 160–177, Feb. 1984.
- [2] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of 24th Design Automation Conference*. ACM/IEEE, 1987, pp. 9–16.
- [3] J. Bhadra, A. K. Martin, and J. A. Abraham, "A Formal Framework for Verification of Embedded Custom Memories of the Motorola MPC7450 Microprocessor," *Formal Methods in Systems Design*, vol. 27, no. 1-2, pp. 67–112, 2005.
- [4] P. Agrawal, "Automatic Modeling of Switch-Level Networks Using Partial Orders," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 7, pp. 696–707, July 1990.
- [5] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
- [6] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J. S. Moore, "Functional Instantiation in First Order Logic," in *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, 1991, pp. 7–26.
- [7] P. Cappalletti, C. Golla, P. Olivo, and E. Zaroni, Eds., *Flash Memories*. Kluwer Academic Publishers, 1999.
- [8] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1990.
- [9] D. Park, "Concurrency and Automata on Infinite Sequences," in *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, ser. LNCS, vol. 104. Springer-Verlag, 1981, pp. 167–183.
- [10] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.
- [11] P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, 2001.
- [12] Z. Barzilai, D. K. Beece, L. M. Hiusman, V. S. Iyengar, and G. M. Silberman, "SLS – a Fast Switch Level Simulator for Verification and Fault Coverage Analysis," in *Proceedings of 23rd Design Automation Conference*, 1986, pp. 164–170.
- [13] N. Krishnamurthy, A. K. Martin, M. S. Abadir, and J. A. Abraham, "Validating PowerPC<sup>TM</sup> Microprocessor Custom Memories," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 61–76, 2000.
- [14] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127.
- [15] A. Salem, "Semi-formal verification of VHDL-AMS descriptions," in *Intl. Symp. on Circuits and Systems*, 2002, pp. 123–127.
- [16] A. Ghosh and R. Vemuri, "Formal Verification of Synthesized Analog Designs," in *Intl. Conf. on Computer Design*, 1999, pp. 40–45.
- [17] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda, "Verification of Analog and Mixed-signal Circuits Using Timed hybrid Petri Nets," in *Automated Technology for Verification and Analysis*, ser. LNCS, no. 3299, 2004, pp. 426–440.
- [18] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena, "Analog Case Study, PROSYD Deliverable D3.4/2," Jan. 2007.

# Author Index

Adams, Sara.....	127	Hanna, Ziyad.....	20
Aziz, Adnan.....	93	He, Yujing.....	111
Babic, Domagoj.....	27	Hoos, Holger H. ....	27
Bhadra, Jayanta.....	239	Hu, Alan J. ....	27
Björk, Magnus.....	127	Huben, Gary A. Van.....	188
Bois, Guy.....	207	Hurst, Aaron P. ....	181
Bradley, Aaron R. ....	173	Hutter, Frank.....	27
Brayton, Robert K. ....	165, 181	Ivancic, Franjo.....	77
Carroll, Hyrum.....	216	Kaiss, Daher.....	20
Case, Michael L. ....	165	Kalyanasundaram, K. ....	69
Cavada, R. ....	69	Khasidashvili, Zurab.....	20
Chechik, Marsha.....	3	Khurshid, Sarfraz.....	93
Chen, Xiaofang.....	53	Klarlund, Nils.....	45
Chen, Yan.....	111	Kroening, Daniel.....	85
Chockler, Hana.....	101	Kupferman, Orna.....	146
Cimatti, A. ....	69	Liffiton, Mark H. ....	13
Claessen, Koen.....	139	Lustig, Yoad.....	146
Clement, Mark.....	216	Mahajan, Yogesh.....	62
Cohen, Ariel.....	37	Malik, Sharad.....	62
Davies, Jessica.....	3	Mangassarian, Hratch.....	13
Eveking, Hans.....	158	Manna, Zohar.....	173
Farchi, Eitan.....	101	Melham, Tom.....	127
Fisler, Kathi.....	154	Mercer, Eric G.....	216
Flaisher, Alon.....	192	Mishchenko, Alan.....	165, 181
Franzen, A. ....	69	Mony, Hari.....	188
German, Steven M. ....	53	Novikov, Sergey.....	101
Ghafari, Naghmeh.....	45	Oberkönig, Martin.....	158
Gluska, Alon.....	192	O'Leary, John W. ....	37
Godlin, Benny.....	101	Pike, Lee.....	231
Gopalakrishnan, Ganesh.....	53	Pnueli, Amir.....	37
Greenstreet, Mark R. ....	199	Ray, Sandip.....	239
Gupta, Aarti.....	77	Roper, Randall J. ....	216
Gurfinkel, Arie.....	3, 45	Roveri, M. ....	69

# Author Index

Rungta, Neha.....	216	Snell, Quinn.....	216
Safarpour, Sean.....	13	Tahar, Sofiene.....	207
Sakallah, Karem A. ....	13	Trefler, Richard.....	45
Sammane, Ghiath Al.....	207	Tuttle, Mark R. ....	37
Schickel, Martin.....	158	Veneris, Andreas.....	13
Schmaltz, Julien.....	223	Wang, Chao.....	77
Seger, Carl-Johan.....	127	Weissenbacher, Georg.....	85
Seigler, Adrian E. ....	188	Xie, Fei.....	111
Shyamasundar, R. K. ....	69	Yan, Chao.....	199
Simmonds, Jocelyn.....	3	Yang, Jin.....	111
Singerman, Eli.....	192	Zaki, Mohamed.....	207
Skaba, Marcelo.....	20	Zaraket, Fadi.....	93
Smith, Edward.....	119	Zuck, Lenore D. ....	37