FMCAD 2009
Formal Methods in Computer Aided Design
November 15 - 18 • Austin, Texas, USA

IEEE Catalog Number: CFP09FMC
ISBN: 978-1-4244-4966-8
Library of Congress: 2009906347

Photography Courtesy
Brian K. Loflin

2009 International Conference • Formal Methods in Computer Aided Design

Armin Biere and Carl Pixley, Editors

9th International Conference
Formal Methods in Computer Aided Design
FMCAD 2009

Austin, Texas
November 15-18, 2009

Proceedings of

9th International Conference

2009 Formal Methods in Computer-Aided Design

FMCAD 2009

15-18 November 2009, Austin, Texas, USA

IEEE Catalog Number: CFP09FMC-CDR
ISBN: 978-1-4244-4966-8
Library of Congress: 2009906347

Technically sponsored by:

IEEE

CEDA
IEEE Council on Electronic Design Automation

In cooperation with:

acm Association for Computing Machinery

sigda

*2009 Formal Methods in Computer-Aided Desig*n

IEEE Catalog Number CFP09FMC-CDR
ISBN 978-1-4244-4966-8
Library of Congress Number 2009906347

Information about how to order the publication, reprint or republication permission, write to:

IEEE Operations Center
445 Hoes Lane
Piscataway, NJ 08854-4150
USA
+1 800 678 IEEE (+1 800 678 4333)
+1 732 981 1393
+1 732 981 9667 (FAX)
email: customer-service@ieee.org

Editorial production by William N. N. Hung
Cover photo courtesy Brian K. Loflin

Technical inquiry email: william_hung@alumni.utexas.net

Table of Contents

Preface

FMCAD 2009, held in Austin, Texas, from November 15th through the 18th, is the ninth 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 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, 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. FMCAD is now a yearly conference.

There were 90 submitted papers, of which 30 were accepted for presentation, 24 as regular and 6 as short papers. There is a wide range of formal topics that were covered in the conference: model checking, software verification, SMT, games, quantitative reasoning, assume/guarantee reasoning, debugging, and synthesis.

We had keynotes from Turing Award winner E. Allen Emerson from the University of Texas talking about "Formal Methods: From Start to Limit" and Intel Vice President John D. Barton talking about post-silicon verification. On Sunday there were tutorials and industrial experience presentations from the following: Moshe Vardi (Rice University) discussed the research opportunities in high-level modeling, design and verification; Nicolaj Bjørner (Microsoft) surveyed the state of Bit-Precise reasoning and word-level SMT; John Penix (Google) discussed the applicability of formal technology to enterprise software; Michael Theobald presented his experience in using formal methods within D. E. Shaw research; Jörg Borman (OneSpin solutions) discussed his concept of total verification.

Monday through Wednesday we had technical sessions on each of the topics listed above. In addition, there were two panels, one of semiconductor companies who are the main consumers of formal tools and another panel of EDA vendors who supply the tools.

We would like to sincerely thank the sponsors of FMCAD'09 for their generous contributions: Intel, IBM. Jasper, NEC, Nvidia, and IEEE. FMCAD'09 is affiliated with IEEE and ACM whom we thank for their support.

We also need to thank the organizing committee: William Hung (publications chair), Sandip Ray and Anna Slobodova (local arrangements co-chairs), Vigyan Singhal (panels chair) and Warren A. Hunt, Jr. (finance chair). They were immensely helpful to the co-chairs to get FMCAD'09 running. We would also like to thank the steering committee (Jason Baumgartner, Aarti Gupta, Warren A. Hunt, Jr., Panagiotis Manolios and Mary Sheeran) for their invaluable advice.

Most of all we would like to thank the FMCAD'09 program committee for their diligent work and their spirited discussions. Without them, the conference would not have the high reputation that it does. We always need to remember that all of these people work without pay to make FMCAD'09 a success. Of course, the conference could not exist without authors choosing to submit their work to critical scrutiny at FMCAD'09.

Armin Biere and Carl Pixley (co-chairs)

FMCAD 2009 Conference Organization

Organizing Committee:

General Chairs:	Armin Biere, *Johannes Kepler University*, Austria Carl Pixley, *Synopsys Inc*., USA
Publications Chair:	William N. N. Hung, *Synopsys Inc*., USA
Local Arrangements:	Sandip Ray, *University of Texas at Austin*, USA Anna Slobodova, *Centaur Technology*, USA
Panel Chair:	Vigyan Singhal, *Oski Technology*, USA

Program Committee:

Jason Baumgartner, *IBM*, USA
Armin Biere, *Johannes Kepler University*, Austria
Per Bjesse, *Synopsys Inc.*, USA
Roderick Bloem, Graz University of Technology, Austria
Gianpiero Cabodi, *Politecnico di Torino*, Italy
Supratik Chakraborty, *IIT Bombay*, India
Alessandro Cimatti, *FBK-IRST*, Italy
Koen Claessen, *Chalmers University of Technology*, Sweden
Leonardo de Moura, *Microsoft*, USA
Wolfgang Ecker, *Infineon*, Germany
Masahiro Fujita, *University of Tokyo*, Japan
Ganesh Gopalakrishnan, *University of Utah*, USA
Aarti Gupta, *NEC Labs America*, USA
Ziyad Hanna, *Jasper*, USA
John Harrison, *Intel*, USA
Alan Hu, *University of British Columbia*, Canada
Warren Hunt, *University of Texas at Austin*, USA
Daher Kaiss, *Intel*, Israel
Daniel Kroening, *University of Oxford*, UK
Thomas Kropf, *Robert Bosch GmbH*, Germany
Andreas Kuehlmann, *Cadence*, USA
Wolfgang Kunz, *University of Kaiserslautern*, Germany
Panagiotis Manolios, *Northeastern University*, USA

Joao Marques-Silva, *University College Dublin*, Ireland
Anmol Mathur, *Calypto*, USA
Ken McMillan, *Cadence*, USA
Tom Melham, *University of Oxford*, UK
Ganapathy Parthasarathy, *Real Intent*, USA
Lee Pike, *Galois*, USA
Carl Pixley, *Synopsys Inc.*, USA
Mukul Prasad, *Fujitsu*, USA
Sandip Ray, *University of Texas at Austin*, USA
Anna Slobodova, *Centaur Technology*, USA
Ofer Strichman, *Technion*, Israel
Helmut Veith, *Technical University of Darmstadt*, Germany
Karen Yorav, IBM, Israel

Steering Committee:

Jason Baumgartner, *IBM*, USA
Aarti Gupta, *NEC Labs America*, USA
Warren Hunt, *University of Texas at Austin*, USA
Panagiotis Manolios, *Northeastern University*, USA
Mary Sheeran, *Chalmers University of Technology*, Sweden

Corporate Sponsors:

IBM Corporation

Intel Corporation

Jasper Design Automation

NEC Labs America

NVIDIA

Additional Reviewers:

Aaron Hurst
Aiguo Xie
Albert Oliveras
Alberto Griggio
Alex Groce
Alexander Kaiser
Alexander Nadel
Alfred Kölbl
Anders Franzen
Andreas Griesmayer
Andreas Holzer
Angelo Brillout
Annette Bunker
Anubhav Gupta
Arie Gurfinkel
Ariel Cohen
Benjamin Meakin
Bhargav Gulavani
Binghao Bao
Christoph Wintersteiger
Cyrille Valentin Artho
Daron Vroon
Djones Lettnin
Dmitry Korchemny
Don Stewart
Donald Chai
Doron Bustan
Duc-Minh Nguyen
Evgeny Pavlenko
Flemming Andersen
Florian Zuleger
Franjo Ivancic
Geert Janssen
Georg Hofferek
Georg Weissenbacher
Gerard Basler
Grzegorz Szubzda
Guodong Li
Hana Chockler
Hari Mony
Harsh Raju Chamarthi
Himanshu Jain
Holger Hermanns
Indradeep Ghosh
Ingo Pill
In-Ho Moon
Jared Davis
Jesse Bingham
Jiazhao Xu
Jie-Hong Jiang
Jim Grundy
Joe Hurd
Johannes Kinder
John Erickson
John Matthews
John Moondanos
Kairong Qian
Karin Greimel
Kevin Harer
Kristin Yvonne Rozier
Leopold Haller
Levent Erkök
Marco Bozzano
Marco Roveri
Mark Kattenbelt
Marko Samer
Markus Wedler
Martin Feldhofer
Martin Fränzle
Max Thalmaier
Michael Case
Michael Tautschnig
Michele Mazzucchi
Mirron Rozanov
Mitra Purandare
Murali Talupur
Nannan He
Nathan Kitchen
Nicolas Blanc
Nikhil Sharma
Nikolaj Bjørner
Ohad Shacham
Pankaj Chauhan
Patricia Bouyer
Peter Dillinger
Philipp Rümmer
Pierre-Alain Reynier
Rachel Tzoref
Robert Brummayer
Robert Kanzelman
Sagar Chaki
Sergio Nocco
Sharon Barner
Shuvendu Lahiri
Sitvanit Ruah
Sivaram Gopalakrishnan
Sol Swords
Sriram Sankaranarayanan
Stefan Lämmermann
Stefano Quer
Stefano Tonetta
Stephen Plaza
Steven German
Subodh Sharma
Sudarshan Srinivasan
Sumit Jha
Tatyana Veksler
Thomas Wahl
Tobias Welp
Verena Wolf
Viktor Schuppan
Yossi Levhari
Ziv Nevo
Zurab Khasidashvili

Monday Keynote

Formal Methods: From Start to Limit

E. Allen Emerson

University of Texas

Abstract

Computer systems exhibit complex and unanticipated behavior. The notion that it might be necessary to mathematically prove that a computer program exhibits the desired behavior can be traced back at least as far as Turing. Today, model checking, a technique based on state space search rather than proof, has shown itself surprisingly useful in routinely performing a range of verification tasks, in particular for hardware verification. The utility of model checking has hinged on the ability to express correctness properties of interest, and the amelioration of state explosion. Even with advances in abstraction and compact symbolic representation, however, there is still a gulf in size between the verifiable systems and numerous practical systems that still exhibit unpredictable behavior. Moreover, the very formality of assertion languages for expressing correctness limits their intuitive understandability, and discourages their broader adoption beyond a band of formal methods experts. The disparity in notation describing the system implementation versus the specification formalism is another complication.

Placed in a larger context, we would argue that the central problem, which is by no means solved, is how to program: how to develop a program (meaning, e.g., software program, hardware design, or embedded system) that exhibits the intended behavior. In this talk we address several aspects of this issue ranging from standard but important verification fare such as improved efficiency. more convenient specification, and the need for more workable divide-and-conquer reasoning to more ambitious strategies such as program synthesis and repair.

Bio

E. Allen Emerson is co-recipient of the 2007 ACM A.M. Turing Award for the invention and development of model checking. His other scientific interests include automata, dynamical systems, and automated program synthesis. He received the BS in Mathematics from the University of Texas, and the PhD in Applied Mathematics from Harvard University. He is now Regents Chair and Professor of Computer Science at the University of Texas.

Wednesday Keynote

Post Silicon Validation/Verification practices in the PC industry

John D. Barton

Intel

Abstract

John will give an overview of Intel Corporation's post silicon validation approach. He will focus on the unique advantages a post silicon approach offers over simulation and emulation based approaches. He will survey the major gaps in fundamental knowledge and tools suggesting potential research or product opportunities.

Bio

John D. Barton is vice president, Digital Enterprise Group, and general manager, Platform Validation Engineering. He is responsible for an organization that performs post-silicon validation/verification for Intel platforms.

Barton's responsibilities include the verification of logically correct operation of microprocessors and chipsets, verification of platform electrical interconnects, software compatibility and security. Barton’s organization also works directly with customers and third parties to ensure key products and technologies ramp with high quality.

Barton joined Intel in 1982 as a software engineer in the Development Systems Operation. He later managed various hardware and software tools for Intel's X86 and 80960 processor families. Barton subsequently managed customer support for Intel's Supercomputer Systems Operation before joining the Intel® Pentium®Pro team as validation manager.

Barton graduated from Oregon State University in 1980 with a bachelor's degree in computer science.

Sunday Tutorial, 15th November, 12:30 - 13:30

Bit-Precise Constraints: Applications and Decision Procedures

Nikolaj Bjørner

Microsoft

Abstract:

Satisfiability Modulo Theories (SMT) is about checking the satisfiability of logical formulas over one or more theories. It draws on a combination of some of the most fundamental areas in computer science as it combines the problem of Boolean satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. It also draws on the most prolific problems in the past century of symbolic logic: the decision problem, completeness and incompleteness of logical theories, and finally complexity theory. The problem of modularly combining special purpose algorithms for each domain is as deep and intriguing as finding new algorithms that work particularly well in the context of a combination. While the basis for SMT solvers is highly inspiring in the context of the intersection of logic, algorithms and systems, it is the role in software and hardware engineering applications that drives the field. Modern software, hardware analysis and model-based tools are increasingly complex and multi-faceted software systems. However, at their core is invariably a component using symbolic logic for describing states and transformations between them. SMT solvers are gaining a distinguished role in this context since they offer support for most domains encountered in programs. A well tuned SMT solver that takes into account the state-of-the-art breakthroughs usually scales orders of magnitude beyond custom ad-hoc solvers. The solver Z3, developed at Microsoft Research, is a state-of-the-art SMT solver. It is largely a product of applications being pursued both in the context of research as well as in Microsoft's product groups. This tutorial will take as starting point applications of Z3 where the bit-precise features are used. These applications include the Dynamic Symbolic Execution tools Pex, available externally for .NET developers, and SAGE, used internally for identifying security vulnerabilities in media format readers. Also included among consumers of bit-precise decision procedures is the model-based testing tool SpecExplorer, and the static program analysis tool PREfix that is used for analyzing Microsoft code repositories containing hundreds of millions lines of code. We give an introduction to decision procedures used (in Z3) for deciding bit-vector constraints and point to the challenges we have encountered for scaling bit-precise decision procedures.

Bio:

Nikolaj Bjørner is a researcher and is currently managing the Foundations of Software Engineering (FSE) group at Microsoft Research in Redmond. FSE's research projects span logics for authentication (Yuri Gurevich), model program analysis (Margus Veanes), parameterized unit test case generation (Niklai Tillmann and Peli de Halleux), model-based design techniques (Ethan Jackson) and together with Leonardo de Moura work around the state-of-the-art Satisfiability Modulo Theories Solver Z3. Previously at Microsoft, Nikolaj developed and shipped the distributed file replication system (DFS-R) in quite different contexts such as Windows Server and Live Messenger, and in a more distant past he developed the Stanford Temporal Prover STeP.

Sunday Tutorial, 15th November, 13:30 - 14:30

Formal Techniques for SoC Verification

Moshe Y. Vardi

Rice University

Abstract:

SystemC has emerged lately as a de facto, open, industry standard modeling language, enabling a wide range of modeling levels, from RTL to system level. Its increasing acceptance is driven by the increasing complexity of designs, pushing designers to higher and higher levels of abstractions. It is particularly suitable for modeling Systems-on-Chip, enabling hardware-software codesign. While a major goal of SystemC is to enable verification at higher level of abstraction, enabling early exploration of system-level designs, the focus so far has been on traditional dynamic verification techniques. It is fair to see that the development of formal-verification techniques for SystemC models is at its infancy. In spite of intensive recent activity in the development of formal-verification techniques for software, extending such techniques to SystemC is a formidable challenge. The difficulty stems from both the object-oriented nature of SystemC, which is fundamental to its modeling philosophy, and its sophisticated event-driven simulation semantics. In this tutorial we portray what is needed to develop formal techniques for SystemC verification, augmenting dynamic validation techniques. By formal techniques we refer here to a range of techniques, including assertion-based dynamic validation, symbolic simulation, formal test generation, explicit-state model checking, and symbolic model checking. We describe recent progress in this area and outline research challenges.

Bio:

Moshe Y. Vardi is the George Professor in Computational Engineering and Director of the Ken Kennedy Institute for Information Technology at Rice University. He chaired the Computer Science Department at Rice University from January 1994 till June 2002. Prior to joining Rice in 1993, he was at the IBM Almaden Research Center, where he managed the Mathematics and Related Computer Science Department. His research interests include database systems, computational-complexity theory, multi-agent systems, and design specification and verification. Vardi received his Ph.D. from the Hebrew University of Jerusalem in 1981. He is the author and co-author of over 350 articles, as well as two books, "Reasoning about Knowledge" and "Finite Model Theory and Its Applications", and the editor of several collections. He is currently the Editor-in-Chief of the Communications of the ACM. Vardi is the recipient of three IBM Outstanding Innovation Awards, a co-winner of the 2000 Goedel Prize, a co-winner of the 2005 ACM Kanellakis Award for Theory and Practice, a co-winner of the 2006 LICS Test-of-Time Award, a co-winner of the 2008 ACM PODS Mendelzon Test-of-Time Award, a winner of the 2008 ACM SIGMOD Codd Innovations Award, a recipient of the 2008 Blaise pascal Medal for Computer Science by the European Academy of Sciences, as well as a 2008 ACM Presidential Award. He holds honorary doctorates from the University of Saarland, Germany, and the University of Orleans, France. Vardi is an editor of several international journals, and Editor-in-Chief of the Communication of ACM. He is Guggenheim Fellow, as well as a Fellow of the Association of Computing Machinery, the American Association for the Advancement of Science, the Association for the Advancement of Artificial Intelligence, and the Institute for Electrical and Electronic Engineers. He was designated Highly Cited Researcher by the Institute for Scientific Information, and was elected as a member of the US National Academy of Engineering, the European Academy of Sciences, and the Academia Europea.

Industrial Experience Report

15th November, 15:00 – 16:00

Formal Verification Challenges of a Supercomputer-Class Machine Designed for Molecular Dynamics

Michael Theobald

D.E. Shaw Research

Abstract

In this talk, I will introduce the formal verification challenges encountered in the design of Anton, a massively parallel, special-purpose machine for molecular dynamics simulations. I will review approaches that have had the most impact on the design verification of the chip, such as bug hunting, root-cause analysis, coverage closure and deadlock detection. A particular outcome of the verification effort for Anton is a method that attempts to identify hard-to-verify logic early in the design cycle, based on model checking. This approach allows early modifications of the RTL code that enhance its verifiability for both formal and simulation methods, reducing the long tail of verification time and effort.

Bio

Michael Theobald designs practical formal verification methods for complex circuits at D.E. Shaw Research. He is involved in the development of a special-purpose machine for molecular dynamics, called Anton. Michael also teaches as an Adjunct Professor in the Computer Science Department at Columbia University. Michael received a Ph.D. from Columbia University in 2002 and a Diplom from Johann Wolfgang Goethe-Universitaet, Frankfurt, both in computer science. Prior to joining D.E. Shaw Research in 2004, he was a Postdoctoral Fellow in the Computer Science Department at Carnegie Mellon University with Edmund Clarke.

Industrial Experience Report

15th November, 16:00 – 17:00

Challenges and Opportunities in Deploying Enterprise-Wide Program Analysis Tools

John Penix

Google

Abstract

Beginning in 2006, a small team within Google began deploying a system to automatically run static analysis tools whenever some part of the codebase changes and notify developers of potential problems. While the tools were able to identify problems in any given version of the code, providing accurate, timely results for a large, rapidly changing code base turned out to be extremely challenging.

I'll present the architecture of our tool automation infrastructure and discuss requirements on how fast and accurate analysis tools need to be to provide value in a large-scale fast-paced software development environment. I'll then describe how the system evolved during our various attempts at making improvements, including:

- Integrating the results into the existing developer workflow,
- Incorporating developer feedback to filter and prioritize warnings,
- Dealing with the many valid reasons that developers have to not care about defects,
- Improving the scalability of the analysis, including some unsolved issues

The general theme is that the experience of the end user makes a big difference if you want people to continue to use your tools over a long period of time. I'll finish with a summary of how various aspects of tool performance impact the end user experience and highlight areas where improvements would be valuable.

Bio

John Penix is a Senior Software Engineer on Google's Engineering Productivity team where he works on large-scale test automation and program analysis platforms. He was the technical lead for an enterprise-wide deployment of static program analysis tools including integration of analysis results into the developer workflow. Prior to joining Google, John was a Computer Scientist at NASA's Ames Research Center where he helped develop techniques to enable model checkers to verify properties of flight software. John received a Ph.D. in Computer Engineering from the University of Cincinnati. He is a member of the Steering Committee for the IEEE/ACM International Conference on Automated Software Engineering.

Industrial Experience Report

15th November, 17:00 – 18:00

Complete Functional Verification

Jörg Bormann

OneSpin Solutions

Abstract

Complete Functional Verification is a practical approach to verify the entire functionality of a circuit. It has been successfully applied in many industrial hardware design projects. This overview presents the technical components of the approach, sketches its scientific foundations and elaborates on the related methodology and its integration into industrial hardware design and verification processes. The approach is based on properties verifying circuit operations like, e.g., a read request of a bus bridge or the instruction execution of a processor pipeline. Such properties are well-suited for SAT based proofs, and for so called "Completeness Checking". The latter proves that a set of properties verifies all circuit functionality. An adaptation of Assume-Guarantee-Reasoning is used to decide when the complete verification of sub-circuits is also a complete verification of the compound circuit. Application examples will illustrate the approach and present circuit sizes and verification efforts.

Bio

Jörg Bormann received a masters degree in Mathematics in 1990. He worked in the formal hardware verification groups of Siemens and Infineon. Since 2005 he has been with OneSpin Solutions GmbH, Munich, where he is now the Director of Advanced Applications and Technology. Jörg has been working on SAT, BDD and theorem prover based formal verification of hardware, both in functional and equivalence verification. He developed Complete Functional Verification including Completeness Checking. For this research Jörg received a PhD in Electrical Engineering from the University of Kaiserslautern, Germany, in 2009. Jörg developed OneSpin's GapFree Verification Process, and defined OneSpin's flagship product 360 MV. Using Complete Functional Verification Jörg conducted several business critical first time right verification projects, e.g., on processors, processor based SOCs and telecom SOCs.

Semiconductor Panel

Frontline users speak up!
What works, What doesn't, and What are they doing about it?

Panelists: Ken Albin, AMD
Alan Carlin, Freescale
Velu Durairaj, TI
Alan Hunter, ARM
Tushar Ringe, ADI
Dan Smith, NVIDIA

Moderator: Adnan Aziz, University of Texas at Austin

EDA Vendors Lunch Panel

What will be the next breakthrough solutions in formal?

Panelists: Harry Foster, Mentor Graphics
Ziyad Hanna, Jasper Design Automation
Kevin Harer, Synopsys
Axel Scherer, Cadence

Moderator: JL Gray, Verilab

Panel Organizer: Vigyan Singhal

Interpolation-Sequence Based Model Checking

Yakir Vizel*†, Orna Grumberg*
*Computer Science Department, The Technion, Haifa, Israel
Email:{yvizel,orna}@cs.technion.ac.il
†Architecture, System Level and Validation Solutions, Intel Development Center, Haifa, Israel

***Abstract*—SAT-based model checking is the most widely used method for verifying industrial designs against their specification. This is due to its ability to handle designs with thousands of state elements and more. The main drawback of using SAT-based model checking is its orientation towards "bug-hunting" rather than full verification of a given specification. Previous works demonstrated how Unbounded Model Checking can be achieved using a SAT solver. In this work we present a novel SAT-based approach to full verification. The approach combines BMC with *interpolation-sequence* in order to imitate BDD-based Symbolic Model Checking. We demonstrate the usefulness of our method by applying it to industrial-size hardware designs from Intel. Our method compares favorably with McMillan's interpolation based model checking algorithm.**

I. Introduction

Model checking [1] is an automatic approach to formally verifying that a given system satisfies a given specification. The system to be verified is modelled as a finite state machine and the specification is described using temporal logic [2]. Model checking algorithms are based on exploration of the models' state space while searching for violations of the specification.

The introduction of BDD-based *Symbolic Model Checking* (SMC) [3] enabled model checking of real-life designs with a few hundreds of state elements. However, current design blocks with well-defined functionality typically have thousands of state elements and more. SAT-based *Bounded Model Checking* (BMC) [4] can handle designs of that scale. However, BMC is limited to finding a counterexample of a bounded length. Thus, BMC is usually used for bug hunting.

In this work we present a novel SAT-based approach to full verification. The approach combines BMC with *interpolation-sequence* [5], [6] in order to imitate BDD-based Symbolic Model Checking. Our method runs BMC iteratively as usual. However, at each iteration k, if the checked formula is unsatisfiable, then a sequence of k interpolants $\{I_1^k, \ldots, I_k^k\}$ is computed. I_j^k over-approximates the set S_j of states, reachable from the initial states in j steps. In the next BMC iteration, the newly obtained interpolant I_j^{k+1} is conjuncted with I_j^k. The result, denoted I_j, is itself an over-approximation of S_j, but a more precise one, since it contains less states which are not in S_j. Thus, I_j can be viewed as a *refinement* of the computed interpolants. Further, I_j is guaranteed to include no violation of the checked property.

The process terminates with either a counterexample produced by BMC, or by reaching a fixpoint, indicating that no more reachable states will be found. In the latter case, since no violation of the formula has been encountered so far, it is guaranteed that the property holds.

We emphasize that the setting of combining BMC with interpolation in order to compute an over-approximation of the set of reachable states seems similar to McMillan's interpolation based model checking algorithm [7]. However, exploiting interpolation-sequence the way we do results in a different traversal of the sets of reachable states, thus may converge faster. Furthermore, our algorithm often requires less calls for BMC. The paper includes a thorough comparison between the two methods, both on the algorithmic level and by running experiments. Our comparison identifies important cases in which our algorithm performs better than the one in [7].

We implemented our algorithm and the one in [7] within Intel's verification tool. All experiments were conducted on models from Intel's next generation Microprocessor designs. The checked properties are real specifications, used to verify those designs. The experiments compare various parameters of the two methods. In all our experiments, when a fixpoint could be reached only at a high bound, our method performed better than [7]. The algorithm in [7], on the other hand, performed better when a fixpoint could be reached at a low bound. In addition, falsified properties always favored our method.

When describing our method we assume a safety property of the form AGq, where q is a propositional formula. This, however, does not restrict its generality since model checking of liveness properties can be reduced to handling safety properties [8]. Further, model checking of safety properties can be reduced to handling properties of the form AGq [9].

A. Related Work

SAT-based *Bounded Model Checking* (BMC) [4] is widely used for the verification of large systems. BMC can usually handle much larger designs than other known methods such as BDD based SMC [3]. However, it is mostly limited to bug finding.

Several works extend BMC for full verification. [10] defines a *Reachability Diameter*, which sets a bound on the number of BMC iterations needed for full verification. This bound, however, is usually hard to compute. Moreover, the bound is very large and therefore the resulting formulas are too large for a SAT solver to handle.

[11] suggests to use *Induction* for full verification. This method uses the BMC check as the induction base. Then, the induction step is checked by checking a second formula. Note that induction works automatically only for simple local

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

properties. For complex properties, the user has to come up with a good inductive invariant. *Proof-Based Abstraction* [12] exploits BMC to determine an abstract model on which BDD-based model checking can be applied. *Interpolation-Based Model Checking* [7] exploits interpolation to compute an over-approximation of the reachable states. The latter work is the closest to ours. We compare the two works in a later section, once the details of the methods are presented.

In this work we use *interpolation-sequence* rather than the usual interpolation. Interpolation-sequence has been introduced and used in [5] and [6].

In [5] it is used for computing an abstract model based on predicate abstraction, for software model checking. In [6] interpolation-sequence is used for software model checking and lazy abstraction. While this work uses the interpolation-sequence to compute over-approximations of reachable states and predicates, the computation considers a specific possible execution of the verified software. Our work, on the other hand, uses the interpolation-sequence to gain information on the entire model. Clearly, the two works use different criteria for convergence.

B. Outline

The rest of the paper is organized as follows. In section II we present some background, including interpolation (II-A), model checking (II-B) and bounded model checking (II-C). Our algorithm is described in section III. In section IV we compare our method to the one of [7]. Section V presents implementation details and our experimental results. Finally, we conclude in section VI.

II. Preliminaries

In this section we present a short description of Interpolation, Model Checking and Bounded Model Checking.

A. Interpolation

Throughout the paper we will denote the value *false* as $\bot$ and the value *true* as $\top$. For a formula X, $\mathcal{L}(X)$ is the set of variables appearing in X. For a set of formulas $\{X_1, \ldots, X_n\}$ we will use $\mathcal{L}(X_1, \ldots, X_n)$ to denote the variables appearing in $X_1, \ldots, X_n$.

Definition 2.1. Let (A, B) be a pair of formulas such that $A \wedge B \equiv \bot$. The *interpolant* for (A, B) is a formula I such that:

- $A \Rightarrow I$.
- $I \wedge B \equiv \bot$.
- $\mathcal{L}(I) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B)$.

A SAT solver is a complete decision procedure that given a set of clauses, determines whether the clause set is *satisfiable* or *unsatisfiable*. A clause set is said to be satisfiable if there exists a *satisfying assignment* such that every clause in the set is evaluated to $\top$. If the clause set is satisfiable then the SAT solver returns a satisfying assignment for it. If it is not satisfiable (unsatisfiable), meaning, it has no satisfying assignment, then modern SAT solvers produce a *proof of unsatisfiability* [12], [13]. An interpolant can be produced out of a proof of unsatisfiability [7].

Definition 2.2. Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$. That is $\bigwedge \Gamma = A_1 \wedge \ldots \wedge A_n$ is unsatisfiable. An **interpolation-sequence** for Γ is a set $\{\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n\}$ such that:

1) $\mathcal{I}_0 \equiv \top$ and $\mathcal{I}_n \equiv \bot$
2) For every $0 \leq j < n$ it holds that $\mathcal{I}_j \wedge A_{j+1} \Rightarrow \mathcal{I}_{j+1}$
3) For every $0 < j < n$ it holds that $\mathcal{L}(\mathcal{I}_j) \subseteq \mathcal{L}(A_1, \ldots, A_j) \cap \mathcal{L}(A_{j+1}, \ldots, A_n)$

Computing an interpolation-sequence for a sequence of formulas is done in the following way: for each $\mathcal{I}_i$, $0 < i < n$, the sequence of formulas is partitioned in a different way such that $\mathcal{I}_i$ is the interpolant for the formulas $A(i) = \bigwedge_{j=1}^{i} A_j$ and $B(i) = \bigwedge_{j=i+1}^{n} A_j$.

Theorem 2.3. *Let $\Gamma = \{A_1, A_2, \ldots, A_n\}$ be a set of formulas such that $\bigwedge \Gamma \equiv \bot$ and let Π be a proof of unsatisfiability for $\bigwedge \Gamma$. For every $1 \leq i < n$ let us define $A(i) = A_1 \wedge \ldots \wedge A_i$ and $B(i) = A_{i+1} \wedge \ldots \wedge A_n$. Let $\mathcal{I}_i$ be the interpolant for the pair $(A(i), B(i))$ extracted using Π then the set $\{\top, \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{n-1}, \bot\}$ is an interpolant sequence for Γ.*

B. Model Checking

Model checking [1] is an automatic approach to formally verifying that a given system satisfies a given specification. The system is modelled by a Kripke structure and the specification is written in *temporal logic*. Determining whether a model satisfies a given specification is based on exploration of the model's state space in a search for violations of the specification.

Definition 2.4. Given a set of atomic propositions AP, a *Kripke structure* M is the quadruple $M = (S, \mathit{INIT}, \mathit{TR}, L)$ where S is a finite set of states, $\mathit{INIT} \subseteq S$ is a set of initial states and $\mathit{TR} \subseteq S \times S$ is a total transition relation. That is, for every $s \in S$ there exists $s' \in S$ such that $(s, s') \in \mathit{TR}$. Finally, $L : S \longrightarrow \mathcal{P}(AP)$ is the labeling function which associates with every state $s \in S$ the set $L(s)$ of atomic propositions that hold in s.

A *path* in a Kripke structure M is a sequence of states $\pi = (s_0, s_1, \ldots)$ such that for all $i > 0$, $s_i \in S$ and $(s_i, s_{i+1}) \in \mathit{TR}$. The length of a path is denoted by $|\pi|$. If π is infinite then $|\pi| = \infty$. If $\pi = (s_0, s_1, \ldots, s_n)$ then $|\pi| = n$. A path is an *initial path* when $s_0 \in \mathit{INIT}$.

A formula in *Linear Temporal Logic* (LTL) [2] is of the form $\mathbf{A}f$ where f is a path formula. A model M satisfies an LTL property $\mathbf{A}f$ if all paths in M satisfy f. If there exists a path not satisfying f, this path is defined to be a *counterexample*.

We consider a subset of LTL properties called *safety* properties since Liveness checking can be achieved by the method presented in [8]. In addition, only safety properties of the form AGq are considered where q is a propositional formula. This

does not restrict the applicability of our results, since safety properties can be verified using *invariance checking* [9].

Given a property AGq, the model checking problem can then be described as exploring the state space of a model M while checking that q holds for all states.

Let M be a model, *Reach* be the set of reachable states and let $f = AGq$ be a property. If for every $s \in$ *Reach*, $L(s) \models q$ then the property holds in M. On the other hand, if there exist a state $s \in$ *Reach* such that $L(s) \models \neg q$ then there exists an initial path $\pi = s_0, s_1, \ldots, s_n$ such that $s_n = s$. The path π is a counterexample for the property f.

We would sometimes like to represent a Kripke structure by means of propositional formulas. In order to do so, we define the set of Boolean state variables, denoted V. Given V where $|V| = n$, a state $s \in S$ is represented by a vector in the set $\{0,1\}^n$ and by that s is a valuation of the state variables in V. A set of states can be represented by a formula over V where the truth assignments represent the states. With abuse of notation we will refer to a formula η over V as a set of states and therefore use the notion $s \in \eta$ for states represented by η. For some variable v, v' is used to denote the value of v after one time unit. The set of these variables is denoted by V'. In the general case V^i is used to denote the variables in V after i time units (thus, $V^0 \equiv V$). Let η be a formula over V^i, the formula $\eta[V^i \leftarrow V^j]$ is identical to η except that for each variable $v \in V$, v^i is replaced with v^j.

C. Bounded Model Checking

Bounded model checking (BMC) [4] is an iterative process for checking properties up to a given bound. Let M be a Kripke structure and $f = AGq$ be the property to be verified. Given a bound k, BMC either finds a counterexample of length k or less for f in M, or concludes that there is no such counterexample. In order to search for a counterexample of length k the following propositional formula is built:

Formula 1. $\varphi^k_M(f) = \mathit{INIT}(V^0) \wedge \mathit{TR}(V^0, V^1) \wedge \mathit{TR}(V^1, V^2) \wedge \ldots \wedge \mathit{TR}(V^{k-1}, V^k) \wedge (\neg q(V^k))$

$\varphi^k_M(f)$ is then passed to a SAT solver which searches for a satisfying assignment. If there exists a satisfying assignment for $\varphi^k_M(f)$ then the property is violated, since there exists a path of length k violating the property. In order to conclude that there is no counterexample of length k or less, BMC iterates all lengths from 0 up to a given threshold bound k. At each iteration a SAT procedure is invoked.

When M and f are obvious from the context we omit them from the formula $\varphi^k_M(f)$ denoting it as φ^k. The BMC algorithm is described in Fig 1.

The main drawback of this approach is the fact that it is not complete. It can only guarantee that there is no counterexample of size smaller or equal to k. It cannot guarantee that there is no counterexample of size greater than k.

```
1: function BMC(M,f,k)
2:     i := 0
3:     while i ≤ k do
4:         build φ^i_M(f)
5:         result = SAT(φ^i_M(f))
6:         if result = true then
7:             return cex      // returning the counterexample
8:         else
9:             i = i + 1
10:        end if
11:    end while
12:    return No cex for bound k
13: end function
```

Fig. 1: Bounded model checking

III. A Novel SAT-based Model Checking Approach

In this section we present our novel SAT-based algorithm for unbounded model checking (UMC). The proposed algorithm explores the state space of the model by means of an over-approximation using BMC and interpolation-sequence. The innovation lies in the way BMC is combined with interpolation-sequence to extract the needed information.

From this point and on, we will use M to denote the Kripke structure representing the model and $f = AGq$ for a propositional formula q, as the property to be verified.

In order to better understand our work and the motivation behind it, we will first review some basic concepts of SMC.

A. Revisiting Symbolic Reachability Analysis

SMC performs *forward reachability analysis* by computing sets of reachable states S_j where j is the number of transitions needed to reach a state in S_j when starting from the initial states. Further, for every $j \geq 1$, $S_j(V) \wedge \mathit{TR}(V, V') \equiv S_{j+1}(V')$. Once S_j is computed, if it contains states violating q, a counterexample of length i is found and returned. Otherwise, if $S_j \subseteq \bigcup_{i=1}^{j-1} S_i$ then a *fixpoint* has been reached, meaning that all reachable states have been found already. If none violate the property then the algorithm concludes that $M \models f$.

The method presented in this section demonstrates how over-approximated sets, similar to S_i in their characteristics, can be extracted from BMC using an interpolation-sequence generated after each iteration of the BMC loop. These sets will be used to gain knowledge about the reachable states even though the sets are actually an over-approximation of the reachable states. Informally, we will use the notion of *fixpoint* when we can conclude that all *reachable* states in the model have been visited. Note that, the interpolation-sequence exists for a bound N only when there is no counterexample of length N. In case a counterexample exists, BMC returns a counterexample and the interpolation-sequence is not needed.

B. Interpolation-Sequence Based Model Checking

Definition 3.1. A *BMC-partitioning* for φ^N is the set $\Gamma = \{A_1, A_2, \ldots, A_{N+1}\}$ of formulas such that $A_1 = \mathit{INIT}(V^0) \wedge \mathit{TR}(V^0, V^1)$, for every $2 \leq i \leq N$ $A_i = \mathit{TR}(V^{i-1}, V^i)$ and $A_{N+1} = \neg q(V^N)$. Note that $\varphi^N = \bigwedge_{i=1}^{N+1} A_i$ $(= \bigwedge \Gamma)$.

For a bound N, consider a BMC formula φ^N and its BMC-partitioning Γ. In case φ^N is unsatisfiable, its interpolation-sequence is denoted by $\bar{I}^N = (I_0^N, I_1^N, \ldots, I_{N+1}^N)$. Note that the BMC-partitioning for φ^N contains $N+1$ elements and therefore the interpolation-sequence contains $N+2$ elements where the first element and the last one are always $\top$ and $\bot$, respectively.

Next, we intuitively explain our method. Consider the formula φ^1 and its BMC-partitioning: A_1, A_2. In case that this formula is unsatisfiable there exists an interpolation-sequence of the form $\bar{I}^1 = (I_0^1 = \top, I_1^1, I_2^1 = \bot)$. By Definition 2.2, $S_1 \subseteq I_1^1$ since $\top \wedge A_1 \Rightarrow I_1^1$. Also, $I_1^1 \wedge \neg q(V^1)$ is unsatisfiable, since $I_1^1 \wedge A_2 \Rightarrow \bot$. Therefore, $I_1^1 \models q$. In the next BMC iteration, consider φ^2 and its BMC-partitioning A_1, A_2, A_3. In case that φ^2 is unsatisfiable, we get $\bar{I}^2 = (\top, I_1^2, I_2^2, \bot)$. Here too, $S_1 \subseteq I_1^2$ and the states reachable from it in one transition are a subset of I_2^2 since $I_1^2 \wedge A_2 \Rightarrow I_2^2$. Also, $S_2 \subseteq I_2^2$ and $I_2^2 \models q$. Let us define the sets $I_1 = I_1^1 \wedge I_1^2$ and $I_2 = I_2^2$. These sets have the following properties, $S_1 \subseteq I_1$, $S_2 \subseteq I_2$, $I_1 \models q$ and $I_2 \models q$. Moreover, $I_1[V^1 \leftarrow V] \wedge TR(V, V') \Rightarrow I_2[V^2 \leftarrow V']$.

In the general case if φ^N is unsatisfiable then for every $1 \leq j \leq N$, $S_j \subseteq I_j^N$. If we now define $I_j = \bigwedge_{k=j}^{N} I_j^k$ then for every $1 \leq j \leq N$ we get:

- $I_j \models q$ since $I_j^j \models q$.
- $I_j \wedge TR(V, V') \Rightarrow I_{j+1}$ since $I_j^k \wedge TR(V^j, V^{j+1}) \Rightarrow I_{j+1}^k$ for every $1 \leq k \leq N$
- $S_j \subseteq I_j$ since $S_j \subseteq I_j^k$ for every $1 \leq k \leq N$.

As a result, the sets $I_1, I_2, \ldots, I_N$ can be used to determine if $M \models f$. Intuitively, the sets I_j are similar to the sets S_j computed by SMC except that they are over-approximations of S_j. Therefore, these sets can be used to imitate the forward reachability analysis of the model's state-space by means of an over-approximation. This is being done in the following manner. BMC runs as usual with one extension. After checking bound N, if a counterexample is found, the algorithm terminates. Otherwise, the interpolation-sequence $\bar{I}^N$ is extracted and the sets I_j for $1 \leq j \leq N$ are updated. If $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ for some $1 \leq j \leq N$, then we conclude that a fixpoint has been reached and all reachable states have been visited. Thus, $M \models f$. If no fixpoint is found, the bound N is increased and the computation is repeated for $N+1$.

Informally, the following facts are needed in order to guarantee the correctness of the algorithm described above for checking $M \models f$. For every $1 \leq j \leq N$ we need:

1) I_j should satisfy q.
2) $I_j(V) \wedge TR(V, V') \Rightarrow I_{j+1}(V')$ for $j \neq N$.
3) $S_j \subseteq I_j$.

This means that the algorithm cannot be implemented using $\bar{I}^N$ alone. This is because $\bar{I}^N$ does not satisfy condition (1): while $I_N^N \models q$, I_j^N for $j \neq N$, does not necessarily satisfy q. This can be remedied by conjuncting each I_j^N with I_j^j. However, now condition (2) no longer holds. Taking $I_j = \bigwedge_{k=j}^{N} I_j^k[V^j \leftarrow V]$ results in a set with all three properties.

```
1: function UPDATEREACHABLE(Ī,Ī^k)
2:     j = 1
3:     while (j < k) do
4:         I_j = I_j ∧ I_j^k
5:         Ī[j] = I_j
6:         j = j + 1
7:     end while
8:     Ī[k] = I_k^k
9: end function
```

Fig. 2: Updating the reachability vector

```
1: function FIXPOINTREACHED(Ī)
2:     j = 2
3:     while (j ≤ Ī.length) do
4:         R = ⋁_{k=1}^{j-1} I_k
5:         φ = I_j ∧ ¬R // Negation of I_j ⇒ R
6:         if (SAT(φ) == false) then return true
7:         end if
8:         j = j + 1
9:     end while
10:    return false
11: end function
```

Fig. 3: Checking if a fixpoint has been reached

Definition 3.2. If no counterexample of length N or less exists in M, then $I_j = \bigwedge_{k=j}^{N} I_j^k[V^j \leftarrow V]$ where I_j^k is the j-th element in the interpolation-sequence extracted for the BMC-partitioning of φ^k. The *reachability vector* is defined to be $\bar{I} = (I_1, I_2, \ldots, I_N)$.

The algorithms for updating the reachablility vector and checking for a fixpoint are described in Fig 2 and Fig 3, respectively. The complete model checking algorithm using the method described above is given in Fig 4.

It is important to note that a call to UPDATEREACHABILITY changes the reachability vector. Therefore, the function FIXPOINTREACHED searches for a fixpoint at any point in $\bar{I}$. Moreover, it is not sufficient to check for inclusion of only the last element of $\bar{I}$. Indeed, if for any $j \leq N$, $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ then all reachable states have been found already. However, the implication $I_N \Rightarrow \bigvee_{i=1}^{N-1} I_i$ might not hold due to additional *unreachable* states in I_N. This is because for all $1 \leq j < N$, I_{j+1} is an approximation of the sets reachable from I_j and not the exact image (That is, $I_j \wedge TR(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$ rather than $I_j \wedge TR(V, V') \equiv I_{j+1}[V \leftarrow V']$).

The following lemmas and definition formalize the above and prove the correctness of the algorithm.

Lemma 3.3. *If M does not have a counterexample of length N, then $S_j \subseteq I_j^N[V^j \leftarrow V]$ for every $1 \leq j \leq N$ and $I_N^N \models q(V^N)$.*

Proof: M does not have a counterexample of length N. Therefore, the formula φ^N is unsatisfiable. Let $\bar{I}^N$ be the interpolation-sequence for the BMC-partitioning of φ^N. By Definition 2.2, for $j = 1$, $\top \wedge \mathit{INIT}(V^0) \wedge \mathit{TR}(V^0, V^1) \Rightarrow I_1^1$. For each $2 \leq j \leq N$, $I_i^N \wedge \mathit{TR}(V^j, V^{j+1}) \Rightarrow I_{j+1}^N$. Hence, $S_j \subseteq I_j^N[V^j \leftarrow V]$. Definition 2.2 also state that $I_N^N \wedge \neg q(V^N) \Rightarrow \bot$ and therefore $I_N^N \models q(V^N)$. ∎

Lemma 3.4. *If M does not have a counterexample of length N or less, then $S_j \subseteq I_j$ and $I_j \models q$ for every $1 \leq j \leq N$.*

Proof: For every $j \leq k \leq N$ by Lemma 3.3 $S_j \subseteq I_j^k$ and $I_j^j \models q$. Since I_j is the conjunction of I_j^k for every $j \leq k \leq N$, $S_j \subseteq I_j$ and $I_j \models q$. ∎

Lemma 3.5. *Let $\bar{I} = (I_1, I_2, \ldots, I_N)$ be the reachability vector. For every $1 \leq j < N$, $I_j \wedge \mathrm{TR}(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$.*

Proof: Definition 3.2 and 2.2 imply that $I_j = \bigwedge_{k=j}^{N} I_j^k[V^j \leftarrow V]$ and that for every $j \leq k \leq N$, $I_{j-1}^k \wedge \mathit{TR}(V^{j-1}, V^j) \Rightarrow I_j^k$ we get $I_j \wedge \mathit{TR}(V, V') \Rightarrow I_{j+1}[V \leftarrow V']$. ∎

Theorem 3.6. *Assume there is no path of length N or less violating f in M. If there exist $1 < j \leq N$ such that $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$, then $M \models f$.*

Proof: By assumption, there is no path in M of length N or less that violates f. We now show that given $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ we can conclude that there is no path of any length violating f. Let $R = \bigvee_{i=1}^{j-1} I_i$. By assumption, $I_j \Rightarrow R$ and by Lemma 3.5, for every $1 \leq i < j$, $I_i(V) \wedge \mathit{TR}(V, V') \Rightarrow I_{i+1}(V')$. Thus, $R(V) \wedge \mathit{TR}(V, V') \Rightarrow R(V')$ (1). Moreover, for every $1 \leq i \leq j$ the formula $I_i \wedge \neg q$ is unsatisfiable (since $I_i \models q$ by Lemma 3.4). Hence, $R \wedge \neg q$ is unsatisfiable (2).

We can show by induction we can show that all reachable states are in $R^* = R \vee \mathit{INIT}$. The base case handles an initial state. This holds trivially by the definition of R^*. Now let us assume it holds for all states reachable in k steps. It should be proved for states reachable in $k+1$ steps. Let s_{k+1} be a state reachable in $k+1$ steps from an initial state. Let $\pi = s_0, s_1, \ldots, s_k, s_{k+1}$ be an initial path to s_{k+1}. By the induction hypothesis $s_k \in R^*$. By the fact that $(s_k, s_{k+1}) \in \mathit{TR}$ and by (1) we can conclude that $s_{k+1} \in R^*$.

By assumption, $\mathit{INIT} \models q$ since there is no path of length N or less violating f. By that and (2), $R^* \models q$. Thus, the set of reachable states satisfy q which implies that $M \models f$. ∎

Lemma 3.7. *Suppose $M \models f$ then there exists a bound N such that $\bar{I} = \{I_1, I_2, \ldots, I_N\}$ and there exists an index $1 < j \leq N$ such that $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$.*

Proof: The set of states S is finite. Let us define $N = j = |S| + 1$. $M \models f$ hence for every $0 \leq k \leq N$, φ^k is unsatisfiable. Thus, the interpolation-sequence $\bar{I}^k$ exists for every $0 \leq k \leq N$ and by that the reachability vector $\bar{I} = \{I_1, I_2, \ldots, I_N\}$ exists. Since $|S| < \infty$ we get $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$. ∎

```
1: function ISB(M,f)
2:     k := 0
3:     result = BMC(M, f, 0)
4:     if (result == cex) then
5:         return cex
6:     end if
7:     Ī = ∅        // the reachability vector
8:     while (true) do
9:         k = k + 1
10:        result = BMC(M, f, k)
11:        if (result == cex) then
12:            return cex
13:        end if
14:        Ī^k = (⊤, I_1^k, ..., I_k^k, ⊥)
15:        UPDATEREACHABLE(Ī,Ī^k)
16:        if (FIXPOINTREACHED(Ī) == true) then
17:            return true
18:        end if
19:    end while
20: end function
```

Fig. 4: ISB Algorithm

Theorem 3.8. *There exists a path π of length N such that π violates f if and only if ISB terminates and returns cex.*

Proof: Assume that the minimal violating path is of length N. For $N-1$ there is no path in M violating f. By Theorem 3.6 we get that for every j such that $1 \leq j < N$, $I_j \Rightarrow \bigvee_{i=1}^{j-1} I_i$ does not hold. Therefore, the algorithm cannot terminate by returning $true$ in the first $N-1$ iterations. When the algorithm reaches the N-th iteration, $BMC(M, f, N)$ will return a counterexample and the algorithm terminates. The other direction is immediate. ∎

Theorem 3.9. *For every model M and a property $f = AGq$ there exists N such that ISB terminates.*

Proof: If $M \models f$ it follows by Lemma 3.7 that the algorithm terminates and returns $true$. If there is a path in M that violates f, it follows by Theorem 3.8 that the algorithm terminates and returns cex. ∎

IV. Comparing Interpolation-Sequence Based MC to Interpolation Based MC

In the previous section we presented a new method for model checking, the Interpolation-Sequence Based MC (*ISB*) which combines BMC and interpolation-sequence. The closest work to this one is the Interpolation Based MC (IB) described in [7]. Thus, a comparison between the two works is imperative. Other SAT-based methods for full verification have been surveyed in the related work section. Moreover, the work

function CHECKREACHABLE(M,f,k)
  $R = M.INIT$ // Initialize R - initial states of M
  if $(BMC(M, f, 1, k) == cex)$ **then**
    return cex
  end if
  $M' = M$
  repeat
    $A = J(V^0) \wedge TR(V^0, V^1)$
    $B = TR(V^1, V^2) \wedge \ldots \wedge TR(V^{k-1}, V^k) \wedge (\bigvee_{j=1}^{k} \neg q(V^j))$
    $J = SAT.getInterpolant(A, B)$
    if $J \subseteq R$ **then**
      return $fixpoint$
    end if
    $R = R \cup J$
    $M'.INIT = J$
  until $(BMC(M', f, 1, k) == cex)$
  return $abort$
end function

Fig. 5: Calculating the reachable states using a specific bound

presented in [14] shows a clear advantage to IB over other known methods for verification. We first describe IB, then, we compare the two methods.

The following definition will help us to better describe the differences between the two methods. Recall that the verified property is of the form $f = AGq$.

Definition 4.1. For a set of states T, T is said to be S_j-*approximation* w.r.t N, where $1 \leq j \leq N$, if the following two conditions hold: $S_j \subseteq T$ and there is no path of length $(N - j)$ or less violating q, starting from a state $s \in T$. It is denoted by $S_j \preceq_N T$.

A. Interpolation Based Model Checking (IB)

In [7] McMillan presents a SAT-based model checking algorithm for full verification by combining BMC and Craig's Interpolation [15]. The interpolant is used to compute an over-approximation of the set of reachable states. The algorithm concludes that the property holds and no counterexample exists when a fixpoint is reached during the computation of reachable states and none of the computed states violate the property.

The formula φ^k is used in BMC to represent a counterexample of length exactly k. This formula can be modified to represent a counterexample of length l for $1 \leq l \leq k$. We denote this formula by $\varphi^{1,k}$. Consider the following partitioning for $\varphi^{1,k}$:

- $A = INIT(V^0) \wedge TR(V^0, V^1)$
- $B = \bigwedge_{i=1}^{k-1} TR(V^i, V^{i+1}) \wedge (\bigvee_{j=1}^{k} \neg q(V^j))$.

Clearly $\varphi^{1,k} \equiv A \wedge B$. Assume that $\varphi^{1,k}$ is unsatisfiable. By the interpolation theorem [15], there exists an interpolant J_1^k that follows Definition 2.1:

- J_1^k is over the variables of $\mathcal{L}(A) \cap \mathcal{L}(B)$, namely, V^1.
- $A \Longrightarrow J_1^k$. By that, $S_1 \subseteq J_1^k$.
- $J_1^k(V_1) \wedge B$ is unsatisfiable. This means that for every $0 \leq i \leq k-1$, there is no path of length $k-1$ or less starting from J_1^k and violating q.

By the above we get that $S_1 \preceq_k J_1^k$. This procedure is iterated by replacing the initial states in M with the computed interpolant J_1^k. BMC is reinvoked with the same parameters for the modified model $M' = (S, J_1^k[V^1 \leftarrow V], TR, L)$. A new interpolant J_2^k is then extracted. J_2^k satisfies $S_2 \preceq_{k+1} J_2^k$. It is important to notice that J_1^k now satisfies $S_1 \preceq_{k+1} J_1^k$ since the BMC run on M' could not find a counterexample of length k starting from a state in J_1^k. In the general case we replace *INIT* with J_i^k and get J_{i+1}^k. For a given bound k, the computation of over-approximated reachable states appears in Fig 5. Note that after L iterations of the main loop in CHECKREACHABLE we get L interpolants and for every $1 \leq i \leq L$, $S_i \preceq_{k+L} J_i^k$. If at any point, a counterexample is found on a modified model, CHECKREACHABLE is reinvoked with $k+1$. Recall that the counterexample has been obtained on an over-approximated set of states and therefore might not represent a real counterexample in the original model. In case that a real counterexample exists, it will be found during the BMC check on the original model M. In [16], the author suggests to use information from CHECKREACHABLE. If the current bound used is k and at the L-th iteration a counterexample is found, the next bound to use is $k+L$ (rather than $k+1$). This is possible since M is known not to have a counterexample of length $k+L-1$. This heuristic is highly depended on the type of property that is being checked. On the one hand, if the property is false, this heuristic indeed results in better performance. On the other hand, for true properties, this approach may hurt performance since a fixpoint could have been found at a lower bound than $k+L$ (e.g. $k+1$).

A complete description of the algorithms for this method appears in [7].

B. Comparing ISB to IB

The sets of reachable states computed by each method are over-approximated and are different in their characteristics. Therefore, determining which one converges faster is not applicable. A few technical differences exist for ISB and IB. First, the formulas used for the interpolants extraction are different. For a given bound N, ISB uses the formula φ^N while IB uses $\varphi^{1,N}$. Note that in practice, for a certain types of properties $\varphi^{N-j,N}$, for a small $j > 0$, can be used [7]. The problem of using values greater than 1 is that then, termination is not guaranteed. Second, the way the interpolants are computed is different. While ISB computes the sets I_j incrementally and refines them after each iteration of BMC as part of the BMC loop, IB recomputes the sets whenever the bound is increased regardless of previous runs using a different BMC call for each iterpolant. ISB can be viewed as an addition to BMC's loop. The addition is the extraction of an interpolation-sequence at each iteration and the check for a fixpoint. Indeed, after N iterations of the BMC loop in ISB, there are N sets of reachable states $I_1, \ldots, I_N$ and $S_j \preceq_N I_j$. On the other hand, IB consists of two nested loops. The outer

SMC	ISB	IB
$\{S_1, \ldots, S_N\}$	$\{I_1, I_2, \ldots, I_N\}$ $S_i \preceq_N I_i$ After checking bounds 1 to N	$\{J_1^1, J_2^1, \ldots, J_N^1\}$ $S_i \preceq_N J_i^1$ N iterations at bound 1 if possible
$\{S_1, \ldots, S_{N+L}\}$	$\{I_1, \ldots, I_L, \ldots, I_{N+L}\}$ $S_i \preceq_{N+L} I_i$ After checking bounds 1 to $N+L$	$\{J_1^N, J_2^N, \ldots, J_L^N\}$ $S_i \preceq_{N+L} J_i^N$, $(1 \leq i \leq L)$ L iterations at bound N if possible

TABLE I: The correlation between the interpolants computed by ISB and IB to the sets computed using SMC

loop iterates through the bounds while the inner loop calculates the sets of reachable states. If the outer loop is at a higher bound, $N > 1$ and the inner loop performs L iterations then there are L sets of states $J_1^N, \ldots, J_L^N$ such that each has the property $S_i \preceq_{N+L} J_i^N$ $(1 \leq i \leq L)$. Table I summarizes the above.

Having said that, clearly IB can compute, at a given bound, as many sets as needed as long as no counterexample is found (not necessarily a real counterexample). On the other hand, for a bound N, ISB can only compute N sets but it does not require recurrent BMC calls for each bound (only one is needed). By that, we can conclude that in cases IB can compute all the needed sets at a low bound it performs better than ISB. However, for examples where the needed sets can only be computed using higher bounds, ISB has an advantage. This fact is reflected in the results.

As was mentioned before, when a counterexample exists the over-approximated sets of reachable states are not needed. For properties that can be falsified there exists a minimal bound N such that for this bound there exists a path that violates the property. Both algorithms have to hit that bound in order to find the counterexample. Here, ISB has a clear advantage over IB. After each BMC run on the original model, IB executes BMC runs on modified models. This means that there are at least two BMC runs for each bound from 1 to $N-1$. Clearly, the second BMC run is more demanding than the inclusion check performed by ISB. In all our experiments, these kind of properties always favored ISB.

V. Implementation Details and Experimental Results

A. Implementation Details

Both algorithms, ISB and IB, were implemented within Intel's verification system using a SAT-based model checker which is based on Intel's in-house SAT solver *Eureka*. The interpolants are being represented by a data-structure similar to an And-Inverter Graph (AIG) and are being simplified and optimized using known methods such as constant propagation and sharing of redundant expressions.

B. Experimental Results

The proposed algorithm has been checked on various models taken from two of Intel's future CPU designs. The characteristics of the checked models appear in Table III. The 136 properties chosen for the experiments were all real safety properties used to verify the correctness of the designs. The cone of influence for the properties contains thousands of state variables and tens of thousands of gates and signals. The properties vary in that some are *true* and some are *false*. During all checks, a timeout of 10,000 seconds has been set. If after the given timeout the property cannot be verified nor falsified, the process terminates. If the process terminates with no conclusive answer (*Verified* or *Falsified*), it reports that the result is *Bounded* with the highest bound at which the property is known to be non-violated. Experiments were conducted on systems with a dual core Xeon 5160 processors (Core 2 micro-architecture) running at 3.0GHz (4MB L2 cache) with 32GB of main memory. Operating system running on the system is Linux SUSE.

Fig 6 and Fig 7 show the runtime in seconds of running the two interpolation based methods. Each point represents a property from the set of chosen properties. The X axis represents runtime for IB while the Y axis represents the runtime using ISB. We can see that the results vary. All

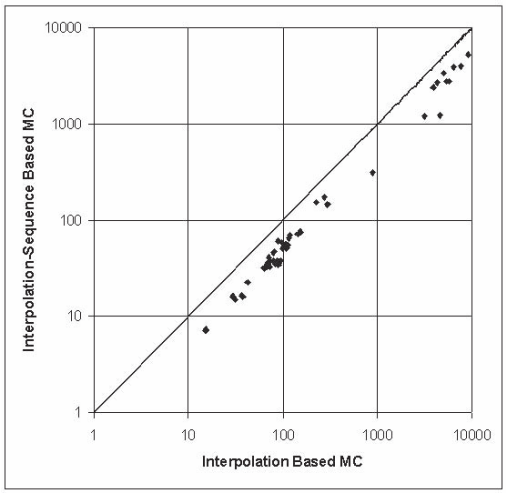


Fig. 6: Runtime of falsified properties from Intel's next micro-architecture releases.

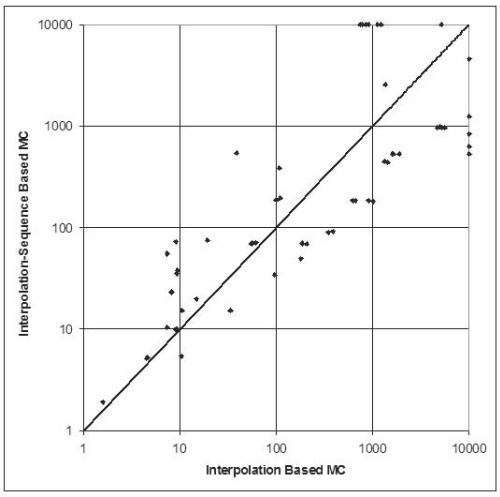


Fig. 7: Runtime of verified properties from Intel's next micro-architecture releases.

Name	♯Vars	B	B_{IB}	♯I	♯I_{IB}	♯BMC	♯BMC$_{IB}$	Time [s]	Time$_{IB}$ [s]
f_1	3406	16	15	136	80	16	80	970	5518
f_2	1753	9	8	45	40	9	40	91	388
f_3	1753	7	6	28	28	7	28	49	179
f_4	1753	16	15	136	94	16	94	473	1901
f_5	3406	6	5	21	13	6	13	68	208
f_6	1761	2	1	3	2	2	2	5	4
f_7	3972	3	1	6	3	3	3	19	14
f_8	2197	3	1	6	3	3	3	10	7
f_9	1629	23	6	276	39	23	39	2544	1340
f_{10}	4894	5	1	15	3	5	3	635	101

TABLE II: Verified properties and their running parameters. ♯ *Vars* stands for the number of state variables in the cone of influence. *B* - bound at convergence, ♯ *I* - number of interpolants computed, ♯BMC - number of calls to BMC algorithm and *Time [s]* - the runtime in seconds

Name	♯ Latches	♯ Inputs	♯ Gates
M_1	3611	3	84570
M_2	4968	2079	133255
M_3	12806	402	89392
M_4	1672	459	11195
M_5	19213	305	146717

TABLE III: Models used for testing

falsified properties (total of 67) favor ISB. Fig 6 shows the runtime for the falsified properties. Fig 7 shows runtime for true properties. There are five properties that can be verified by ISB and not by IB (due to timeout) and two properties that can be falsified using ISB while cannot be falsified using IB. On the other hand, there are seven properties that cannot be verified by ISB but can be verified by IB. The rest of the properties (57 total) are all verified by both algorithms.

A more accurate analysis of the algorithms is shown in Table II that presents running parameters (number of state variables in the cone of influence, bound at convergence, number of interpolants computed, number of calls to BMC and runtime) on various properties for both IB and ISB. For some cases, even though IB converges at a lower bound, and computes less interpolants than ISB, ISB still converges faster by means of runtime. This is due to the fact that BMC calls are computationally heavier than the extraction of the interpolants. Since IB issues more calls to BMC than ISB in these cases, the influence on its runtime is noticeable. Through all our experiments, when convergence for IB could be achieved only at high bounds, ISB always performed better while for convergence at lower bounds, IB is the better performer. This result is supported by the analysis presented in the previous section.

The overall performance, when summarized, are in favor for ISB with 30% improvement in runtime. The total runtime for ISB was 128491 seconds while for IB it was 168745 seconds.

VI. Conclusion

We presented a method that uses interpolation-sequence for SAT-based unbounded model checking. Unlike the interpolation-based model checking algorithm presented in [7], our method does not require successive BMC runs in order to compute an over-approximation of the reachable states. Instead, it is part of the original BMC loop with the addition of interpolation-sequence extraction. It uses a single BMC run for a given bound N to extract information about the reachable states after N transitions or less. The experiments show a clear advantage to ISB when the properties are falsified. In case of true properties, the results vary such that some favor our method while others favor the method of [7]. The overall performance favored our algorithm.

Further investigation can be made in order to characterize the type of properties (when the properties are true) suitable for each method and by that obtain a better understanding of the difference between the two methods. In addition, we believe that the over-approximated sets of reachable states computed using our method at the N-th iteration can be used to simplify the BMC run for bound $N + 1$.

Acknowledgment

The authors would like to thank Ranan Fraer and Yael Meller for their help in conducting this work.

References

[1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT press, 1999.
[2] A. Pnueli, "The temporal logic of programs," in *FOCS'77*.
[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142–170, 1992.
[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking Without BDDs," in *TACAS*, 1999, pp. 193–207.
[5] R. Jhala and K. L. McMillan, "Interpolant-Based Transition Relation Approximation," in *CAV*, 2005, pp. 39–51.
[6] K. L. McMillan, "Lazy Abstraction with Interpolants," in *CAV*, 2006, pp. 123–136.
[7] ——, "Interpolation and SAT-based Model Checking," in *CAV*, 2003, pp. 1–13.
[8] A. Biere and C. Artho, "Liveness checking as safety checking," in *In FMICS: Formal Methods for Industrial Critical Systems, volume 66(2) of ENTCS*, 2002.
[9] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," in *CAV*, 1999, pp. 172–183.
[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, *Bounded model checking*, 2003, vol. 58, pp. 118–149.
[11] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *FMCAD*, 2000, pp. 108–125.
[12] K. L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples," in *TACAS*, 2003, pp. 2–17.
[13] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *DATE*, 2003, pp. 10 880–10 885.
[14] 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, pp. 254–268.
[15] W. Craig, "Linear reasoning. a new form of the herbrand-gentzen theorem," *J. Symb. Log.*, vol. 22, no. 3, 1957.
[16] J. P. M. Silva, "Improvements to the implementation of interpolant-based model checking," in *CHARME*, 2005, pp. 367–370.

Structure-Aware Computation of Predicate Abstraction

Alessandro Cimatti*, Jori Dubrovin†, Tommi Junttila†, Marco Roveri*
*FBK-irst, Embedded Systems Unit, Via Sommarive 18, I-38123 Povo, Trento, Italy
†Helsinki University of Technology TKK, P.O. Box 5400, FI-02015 TKK, Finland

***Abstract*—The precise computation of abstractions is a bottleneck in many approaches to CEGAR-based verification. In this paper, we propose a novel approach, based on the use of structural information.**

Rather than computing the abstraction as a single, monolithic quantification, we provide a *structure-aware* abstraction algorithm, based on two complementary steps. The first, high-level step exploits the structure of the system, and partitions the abstraction problem into the combination of several smaller abstraction problems. This is represented as a formula with quantifiers. The second, low-level step exploits the structure of the formula, in particular the occurrence of variables within the quantifiers, and applies a set of low-level rewriting rules aiming at further reducing the scope of quantifiers.

We experimentally evaluate the approach on a substantial set of benchmarks, and show significant speed ups compared to monolithic abstraction algorithms.

I. Introduction

Many recent approaches to verification are based on Counter-Example Guided Abstraction Refinement (CEGAR). The idea is that the concrete system is overapproximated by an abstract system, that can hopefully be analyzed more easily. If the abstract system is safe, then so is the concrete system. Otherwise, the counterexample witnessing the violation in the abstract space is mapped back to the concrete space. If this operation succeeds, then there is evidence that the property is violated in the concrete space; if not, the abstract counterexample is spurious, and it can be analyzed in order to obtain indications on how to refine the abstraction. The CEGAR framework, originally proposed in the purely Boolean setting [1], has more recently been extended to the case of system representable in more expressive theories [2], [3].

Predicate abstraction is a way to guarantee conservativeness by construction [4]; basically, each predicate characterizes a set of concrete states; the abstract space is generated by collapsing in the same abstract state all the concrete states that share the same evaluation of the predicates. Some recent approaches obtained increased performance by leveraging the power of Satisfiability Modulo Theory (SMT) technologies [5], [6], [7]. Unfortunately, the computation of predicate abstractions remains a bottleneck.

Several CEGAR loops try to mitigate the problem by using imprecise predicate abstractions [2], where the transition relation of the abstract system contains spurious transitions. Although this approach eases the cost of computing the abstraction, it often results in additional CEGAR loop iterations, whose only purpose is to rule out the imprecision of the abstraction.

In this paper, we take on the problem of computing exact predicate abstractions efficiently. We consider that the methods for generating a precise predicate abstraction [5], [6], [7] are monolithic, i.e. start from a symbolic encoding, without taking into account any available structure, e.g. the possible partitioning of the concrete transition relation, the structure of predicates, or the scope of variables. We propose a method to compute predicate abstractions that is aware of, and exploits, the available structural information, thus following a divide-and-conquer approach.

The method proceeds at two different levels. At a *higher level*, we assume that the concrete system to be abstracted is described in a structured language. We discuss basic simplification principles to partition the construction of the abstraction. We instantiate the technique to the case of hybrid automata, that includes partitioning based on asynchrony, scoping of variables, handshake synchronization, and global synchronization (with timed transitions). These simplifications exploit the frame conditions (i.e. that certain variables are not modified), and the cone of influence of the predicates, trying to reduce the number of quantified variables. The technique works on the formula of the transition relation, not committing to a single formalism for models.

At a *lower level*, we transform the quantification tree by means of syntactic transformations aiming at reducing the scope of quantifications even further. This set of reductions exploits the structure of the formula to be quantified (rather than the structure of the models), and can strengthen the partitioning resulting from the high level analysis, by detecting further simplifications that are not readily visible from the structure of the input. The description of the abstract space resulting from the transformations is also structured (e.g. disjunctively partitioned), which can be used to optimize the search in the abstract space. From the technical point of view, there are several ideas that contribute to the efficiency of the approach. First, we perform iterative steps in inlining of equalities and values to simplify the formula to be quantified, and to precompute as much as possible the values of abstract variables. The second idea is to aggressively block partial results from one quantification to the next, thus focusing the search on the unexplored parts of the abstract transition relation. The third and perhaps more interesting idea, called "variable sampling", tries to split a monolithic quantification

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

even further. Variable sampling applies to the case where individual variables would be uncorrelated, were it not for the presence of a single variable: the idea is then to iteratively pick suitable values for the variable to be sampled, and to solve the resulting quantification, that can now be split into separated partitions, and separately quantified.

We implemented the proposed techniques within the NuSMV model checker [8], and we carried out an extensive experimental evaluation on benchmarks from several sources. The results show that the idea of partitioning the computation of predicate abstractions does pay off, and the proposed optimizations are effective in partitioning and in reducing the overall computation time.

The paper is structured as follows. In Sec. II, we discuss some background. In Sec. III, we present the high level structural abstractions. In Sec. IV, we discuss the low level structural abstraction algorithms. In Sec. V, we discuss some relevant related works. In Sec. VI, we report on the experimental evaluation of the approach. Finally, in Sec. VII, we draw some conclusions and outline directions for future work.

II. Background

In this section, we first define our reference formalism, based on Linear Hybrid Automata (Sec. II-A) [9], [10]. Then, in Sec. II-B, we discuss how Linear Hybrid Automata (LHA) can be represented symbolically, in a formalism that is amenable for SMT reasoning. In Sec. II-C, we define the problem of predicate abstraction for LHA. We remark that LHA are chosen as a paradigmatic formalism, generic and expressive, featuring asynchrony, synchronization, time and data variables with frame conditions. However, the results are not limited to this formalism.

A. Linear Hybrid Automata

The following definitions are based on [11], [12], [13]. A *linear atom* over a vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of real-valued variables is an (in)equality of the form $\sum_{1 \leq i \leq n} c_i \cdot x_i \bowtie d$, where $c_1, \ldots, c_n, d \in \mathbb{Q}$ and $\bowtie \in \{<, \leq, =\}$. A *linear predicate* is a finite Boolean combination of linear atoms; a *convex linear predicate* is a finite conjunction of linear atoms. Given a vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of variables, we write $\vec{X}'$ for $\langle x'_1, \ldots, x'_n \rangle$ and, if ϕ is an atom, predicate, or formula over $\vec{X}$, then ϕ' is obtained from ϕ by substituting each variable occurrence x_i with x'_i. If x is a variable, we denote with $\dot{x}$ the variable representing the first derivative of x w.r.t. time elapse.

A *linear hybrid automaton* is a tuple $H = \langle L, \vec{X}, \Sigma, T, Inv, Flow, Init \rangle$ consisting of the following components.

- A finite set L of *control locations*.
- A finite vector $\vec{X} = \langle x_1, \ldots, x_n \rangle$ of real-valued *data variables*.
- A finite set Σ of *synchronization labels*, not including the symbol τ for non-synchronizing transitions.
- A finite set T of *transitions*. Each transition $t \in T$ is a tuple $\langle l, \sigma, act, l' \rangle$, where $l \in L$ is the *source* location, $\sigma \in \Sigma \cup \{\tau\}$ is the *label*, act is the *action*, and $l' \in L$ is the *target* location of t. An action is a pair $act = \langle \vec{Y}, \alpha \rangle$, where $\vec{Y} \subseteq \vec{X}$ is the subset of variables that is updated when t is executed, and the linear predicate α over $\vec{X} \cup \vec{Y}'$ relates the values of the updated variables in the next state with the current values of the variables. The *closure* of act, denoted by $Clos(act)$, is the linear predicate over $\vec{X} \cup \vec{X}'$ obtained from α by adding the frame axioms for the variables that are not updated, i.e. $Clos(act) := \alpha \wedge \bigwedge_{x \in \vec{X} \setminus \vec{Y}} (x' = x)$.
- A mapping Inv from each location $l \in L$ to the *location invariant* $Inv(l)$ that is a convex linear predicate over $\vec{X}$.
- A function $Flow$ mapping each location $l \in L$ to a convex linear *flow* predicate $Flow(l)$ over $\dot{\vec{X}} = \langle \dot{x}_1, \ldots, \dot{x}_n \rangle$. The predicate $Flow(l)$ defines the allowed change derivatives for the variable values when time elapses but the automaton does not perform any discrete transition. Given a predicate $Flow(l)$, we define the predicate $Flow^\star(l)$ over $\vec{X} \cup \{\delta\} \cup \vec{X}'$, where δ is a real-valued variable, by substituting each linear atom $\sum_{1 \leq i \leq n} c_i \cdot \dot{x}_i \bowtie d$ in $Flow(l)$ by $\sum_{1 \leq i \leq n} c_i \cdot (x'_i - x_i) \bowtie d \cdot \delta$. This predicate relates the current variable values with the values after the time has elapsed the amount δ.
- The initial configuration $Init = \langle l^0, \beta \rangle$, where $l^0 \in L$ and β is a convex linear predicate over $\vec{X}$.

A *state* of the automaton H is a pair $s = \langle l, \vec{v} \rangle$, where $l \in L$ is the current control location and $\vec{v} = \langle v_1, ..., v_n \rangle \in \mathbb{R}^n$ associates each data variable x_i with a value v_i. The set of all states is denoted by S. A state $\langle l, \vec{v} \rangle$ is initial if (i) $l = l^0$ and (ii) both $\beta(\vec{v})$ and $Inv(l)(\vec{v})$ evaluate to true. The behavior of H is defined by the *transition relation* $\rightarrow_H \subseteq S \times S$ such that $\langle l_A, \vec{v}_A \rangle \rightarrow_H \langle l_B, \vec{v}_B \rangle$ if and only if (i) $Inv(l_A)(\vec{v}_A)$ and $Inv(l_B)(\vec{v}_B)$ hold, and (ii) either

- (*time elapse step*) (i) $l_B = l_A$, and (ii) $Flow^\star(l)(\vec{v}_A, \delta, \vec{v}_B)$ holds for some $\delta \geq 0$, or
- (*discrete step*) there is a transition $t = \langle l_A, \sigma, act, l_B \rangle \in T$ such that $Clos(act)(\vec{v}_A, \vec{v}_B)$ holds.

Given two hybrid automata, $H_1 = \langle L_1, \vec{X}, \Sigma_1, T_1, Inv_1, Flow_1, Init_1 \rangle$ and $H_2 = \langle L_2, \vec{X}, \Sigma_2, T_2, Inv_2, Flow_2, Init_2 \rangle$, over a common set $\vec{X}$ of variables, their *parallel composition* is the hybrid automaton

$$H_1 \otimes H_2 = \langle L_1 \times L_2, \vec{X}, \Sigma_1 \cup \Sigma_2, T, Inv, Flow, Init \rangle$$

such that

- $\langle \langle l_1, l_2 \rangle, \sigma, act, \langle l'_1, l'_2 \rangle \rangle \in T$ if and only if
  1) $\langle l_1, \sigma, act_1, l'_1 \rangle \in T_1$, $\sigma \notin \Sigma_2$, $l_2 = l'_2$, and $act = act_1$, meaning that if the automaton H_1 executes a σ-transition such that σ is either τ or not in the synchronization alphabet of H_2, then H_2 does nothing;
  2) $\langle l_2, \sigma, act_2, l'_2 \rangle \in T_2$, $\sigma \notin \Sigma_1$, $l_1 = l'_1$, and $act = act_2$, i.e. the case symmetric to the previous one;
  3) $\sigma \neq \tau$, $\langle l_1, \sigma, act_1, l'_1 \rangle \in T_1$ with $act_1 = \langle \vec{Y}_1, \alpha_1 \rangle$, $\langle l_2, \sigma, act_2, l'_2 \rangle \in T_2$ with $act_2 = \langle \vec{Y}_2, \alpha_2 \rangle$, and

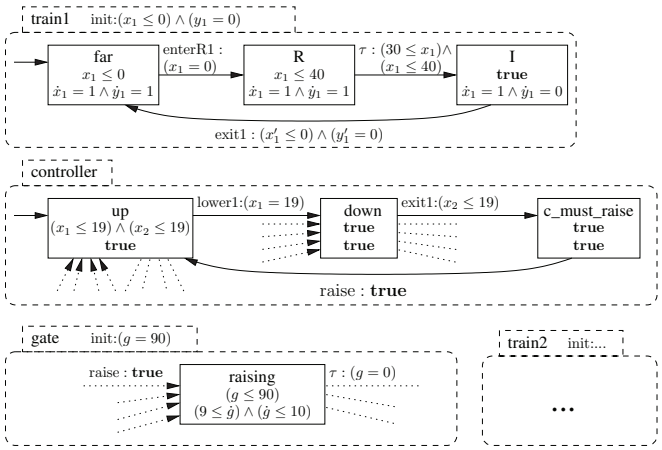


Fig. 1. A (partial) network of linear hybrid automata.

$act = \langle \vec{Y}_1 \cup \vec{Y}_2, \alpha_1 \wedge \alpha_2 \rangle$, meaning that both automata execute a σ-transition synchronously.

- $Inv(\langle l_1, l_2 \rangle) = Inv_1(l_1) \wedge Inv_2(l_2)$ for each $\langle l_1, l_2 \rangle \in L_1 \times L_2$.
- $Flow(\langle l_1, l_2 \rangle) = Flow_1(l_1) \wedge Flow_2(l_2)$ for each $\langle l_1, l_2 \rangle \in L_1 \times L_2$.
- $Init = \langle \langle l_1^0, l_2^0 \rangle, \beta_1 \wedge \beta_2 \rangle$ when $Init_1 = \langle l_1^0, \beta_1 \rangle$ and $Init_2 = \langle l_2^0, \beta_2 \rangle$.

As an example, consider the hybrid automata network "grcver" (taken from the HyTech distribution [14]) partially shown in Fig. 1. The location "far" in the automaton "train1" has the location invariant $(x_1 \le 0)$ and the flow predicate $(\dot{x}_1 = 1) \wedge (\dot{y}_1 = 1)$. In the state in which "train1" is in location "I", "controller" is in "down" and the variable x_2 has value 10, the two automata can synchronize with the label "exit1" and move to locations "far" and "c_must_raise", respectively; the other automata, which are not synchronizing with the label "exit1", do nothing.

B. Symbolic Representation

In order to apply symbolic model checking to a finite network $H_1, ..., H_k$ of linear hybrid automata, we represent the transition relation $\rightarrow_H$ for the composite automaton $H = H_1 \otimes \ldots \otimes H_k$ symbolically without explicitly constructing H. First, assume that each component automaton is of form $H_a = \langle L_a, \vec{X}, \Sigma_a, T_a, Inv_a, Flow_a, Init_a \rangle$ and define the set of *encoding variables* as $\vec{V} := \vec{X} \cup \{loc_1, ..., loc_k\} \cup \Lambda$, where each loc_a is a variable with the domain L_a used to represent the current location of the automaton H_a, and Λ is a set of *auxiliary encoding variables* introduced to encode whether a synchronizing transition is executed or not (see later for details). Now the *symbolic transition relation* encoding for H is a formula R over $\vec{V} \cup \{\delta\} \cup \vec{V}'$ such that a valuation ρ for $\vec{V} \cup \{\delta\} \cup \vec{V}'$ evaluates R to true iff the transition relation $\rightarrow_H$ has a step from the state $\langle \langle \rho(loc_1), \ldots, \rho(loc_k) \rangle, \langle \rho(x_1), ..., \rho(x_n) \rangle \rangle$ to the state $\langle \langle \rho(loc'_1), \ldots, \rho(loc'_k) \rangle, \langle \rho(x'_1), ..., \rho(x'_n) \rangle \rangle$. We define one such R by considering τ-transitions, synchronizing transitions, and time elapse steps separately. To simplify the presentation, we use $\Sigma_\cup$ to denote the set $\cup_{a=1}^k \Sigma_a$ of all synchronization labels and $T_{a,\sigma} = \{t \in T_a \mid t = \langle l, \sigma, act, l' \rangle\}$, where $\sigma \in \Sigma_a \cup \{\tau\}$, for the set of σ-transitions in H_a.

a) Local non-synchronizing transitions: For each component automaton H_a and for each τ-transition $t = \langle l, \tau, act, l' \rangle \in T_{a,\tau}$ in H_a, we define the encoding

$$E_t \;:=\; (loc_a = l) \wedge Clos(act) \wedge (loc'_a = l') \wedge \bigwedge_{1 \le j \le k, j \ne a} (loc'_j = loc_j). \quad (1)$$

capturing the effect of H_a executing t and stating that the other automata do nothing.

b) Synchronizing transitions: The discrete steps corresponding to the execution of synchronizing transitions in several automata is perhaps the most cumbersome to encode. Let $\sigma \in \Sigma_\cup$ be a synchronization label. Let $J_\sigma = \{a \in \{1, \ldots, k\} \mid \sigma \in \Sigma_a\}$ be the set of indices of automata in whose label set σ occurs and define the complement of J_σ by $\bar{J}_\sigma = \{1, \ldots, k\} \setminus J_\sigma$.

First, the set of encoding variables is extended by having the set Λ of auxiliary encoding variables to include a variable $Fire_{a,\sigma}$ with the domain $T_{a,\sigma}$ for each component automaton H_a. Intuitively, $Fire_{a,\sigma} = t$ if the σ-transition t is executed.

Given a σ-transition $t = \langle l, \sigma, act, l' \rangle$ with $act = \langle \vec{Y}, \alpha \rangle$ in an automaton H_a, let $Updated(t) = \vec{Y}$ and define

$$\Psi_{a,t} \;:=\; (Fire_{a,\sigma} = t) \Rightarrow (loc_a = l) \wedge \alpha \wedge (loc'_a = l'). \quad (2)$$

In addition, we define a predicate $Frame_{x,\sigma}$ that evaluates to true iff the frame conditions for the variable x are met, i.e. either some σ-transition that updates x is fired or the variable keeps its current value. Formally,

$$Frame_{x,\sigma} := (x' = x) \vee \bigvee_{a \in J_\sigma} \bigvee_{t \in T_{a,\sigma} \wedge x \in Updated(t)} (Fire_{a,\sigma} = t). \quad (3)$$

Note that if no σ-transition updates x, then $Frame_{x,\sigma} := (x' = x)$. Now define the actual encoding by

$$E_\sigma := \Big(\bigwedge_{j \in \bar{J}_\sigma} (loc'_j = loc_j)\Big) \wedge \Big(\bigwedge_{x \in \vec{X}} Frame_{x,\sigma}\Big) \wedge \Big(\bigwedge_{a \in J_\sigma} \bigwedge_{t \in T_{a,\sigma}} \Psi_{a,t}\Big) \quad (4)$$

c) Time elapse steps: In time elapse steps the component automata do not change their current locations but only time passes in a way that respects the flow predicates of the current locations of the automata. This can be captured by the encoding

$$E_\delta := (\delta > 0) \wedge \bigwedge_{1 \le a \le k} \Big((loc'_a = loc_a) \wedge \bigwedge_{l \in L_a} \Psi_{a,l} \Big) \quad (5)$$

where $\Psi_{a,l} := \big((loc_a = l) \Rightarrow Flow^\star_a(l)\big)$.

d) Putting it all together: We also define *location invariant constraint*

$$C_{\text{Inv}} := \bigwedge_{1 \le a \le k} \bigwedge_{l \in L_a} \big((loc_a = l) \Rightarrow Inv_a(l)\big) \quad (6)$$

that ensures that the location invariants of all automata hold in the current state.

Building on the components described above, we can define a compact *symbolic transition relation* for the composite automaton $H_1 \otimes \ldots \otimes H_k$ as

$$R := \Big(\bigvee_{1 \le a \le k} \bigvee_{t \in T_{a,\tau}} E_t \vee \bigvee_{\sigma \in \Sigma_\cup} E_\sigma \vee E_\delta \Big) \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \quad (7)$$

C. Computing Predicate Abstractions

Let a set $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ of predicates be given. Γ induces a partition on the set of states of the concrete system, each partition containing all the states that assign the same truth values to each predicate. Predicate abstraction defines an abstract system having as states the (finite) set of truth assignments to Γ; a transition between two abstract states a_i and a_j is possible iff there exist two concrete states s_i and s_j such that the evaluation of the predicates in s_i is a_i, the evaluation in s_j is a_j, and $s_i \rightarrow_H s_j$.

Predicate abstraction can be symbolically represented by associating to each predicate a corresponding Boolean variable. Based on the corresponding vector $\vec{P} = \langle p_1, \ldots, p_m \rangle$, we define the *abstraction constraint*

$$C_\Gamma := \bigwedge_{1 \le j \le m} (p_j \Leftrightarrow \gamma_j) \wedge (p'_j \Leftrightarrow \gamma'_j). \quad (8)$$

The *abstract transition relation* of the system obtained by applying predicate abstraction to the given system is symbolically represented by a Boolean formula $\tilde{R}$ over $\vec{P} \cup \vec{P}'$. The formula is equivalent to the following definition:

$$\tilde{R} := \exists \vec{V}, \delta, \vec{V}' : R \wedge C_\Gamma. \quad (9)$$

Similar considerations apply to the symbolic representation of the initial states, and to the abstraction of the error states. How to efficiently compute a quantifier-free Boolean presentation of the above formula is the subject of the next sections.

III. High-level Structural Abstraction

Our goal is to compute a quantifier-free presentation for the *abstract transition relation* $\tilde{R}$ over $\vec{P} \cup \vec{P}'$ defined by

$$\tilde{R} := \exists \vec{V}, \delta, \vec{V}' : R \wedge C_\Gamma.$$

Because existential quantification does not distribute over the conjunctions in (9) and (7), the computation of $\tilde{R}$ can be very expensive. However, when we push the invariant and abstraction constraints in, and distribute the quantifier over the resulting disjunctions, we obtain the *disjunctive abstract transition relation* formulation

$$\begin{aligned} \tilde{R}_{\text{disj}} := \bigvee_{1 \le a \le k} \bigvee_{t \in T_{a,\tau}} \Big(\exists \vec{V}, \delta, \vec{V}' : E_t \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big) \vee \\ \bigvee_{\sigma \in \Sigma_\cup} \Big(\exists \vec{V}, \delta, \vec{V}' : E_\sigma \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big) \vee \\ \Big(\exists \vec{V}, \delta, \vec{V}' : E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \Big). \end{aligned} \quad (10)$$

While increasing the total formula size, this enables us to decompose the computation of the abstract transition relation into smaller computations. Moreover, it enables the application of simplifications as the sub-problems have a simpler structure. For instance, the disjunct encoding a local, non-synchronizing τ-transition $t = \langle l, \tau, act, l' \rangle \in T_{a,\tau}$ of a component automaton H_a is now of form

$$\begin{aligned} \exists \vec{V}, \delta, \vec{V}' : (loc_a = l) \wedge \alpha \wedge \\ \textstyle\bigwedge_{x \in \vec{X} \setminus \vec{Y}} (x' = x) \wedge (loc'_a = l') \wedge \\ \textstyle\bigwedge_{1 \le j \le k, j \ne a} (loc'_j = loc_j) \wedge \\ C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma \end{aligned} \quad (11)$$

where $\vec{Y} \subseteq \vec{X}$ is the set of data variables updated by the action act of the transition t. This formulation enables for the use of the equalities to eliminate the quantified variables loc_a, loc'_a, loc'_j for each $j \ne a$ and x' for each $x \in \vec{X} \setminus \vec{Y}$, by applying inlining (see Sec. IV-A).

Furthermore, this structure-based disjunctive formulation has the benefit of making the formula-based techniques described in the next section more efficient because the formulas bound by the quantifiers have less constraints and the constraints reflect the structure and locality of transitions. For instance, consider the system in Fig. 1 and assume that the set of abstraction predicates Γ is such that each predicate involves variables used either (i) only in the automaton "gate" or (ii) only in other automata. Now the computation of, e.g., the abstraction of the τ-transition in the automaton "train1" is partitioned into two parts automatically: (i) the effect of firing the transition w.r.t. the automata "train1", "train2", and "controller", and (ii) the effect w.r.t. "gate" (which after inlining reduces effectively to stating that "gate" stays in the same location).

Note that although structure-based disjunctive partitioning enhances locality and structure exploitation, there are two aspects that reduce transition and variable locality and thus partly necessitate the formula-level techniques in Sec. IV. First, location invariants of other component automata can refer to variables owned by an automaton and thus control its behavior. For instance, the enabledness of the τ-transition from location "R" of automaton "train1" is indirectly controlled by the automaton "controller" via the variable x_1: when "controller" is in location "up", the transition cannot be enabled. Second, the predicates can "globalize" a local variable if the variable is correlated with (compared with or assigned from/to, either directly or transitively) any non-local variable. When variables are correlated with others, the part of the CEGAR loop that deduces new predicates can introduce predicates that mix the variables, even if they are local to different automata. As an example, in Fig. 1 the variable g is local to the automaton "gate" and none of the transitions, location invariants, or flow predicates that refer to g refers to any other variable. But the time elapse steps implicitly connect the flow predicates of the automata "gate" and "train1" via the variable δ, and after executing the CEGAR loop few times a predicate "$\gamma_i := (9x_1 + g \le 261)$" can be found and added to the set of predicates. Because of this predicate, the abstraction of local transitions of "train1" cannot disregard the variables and location invariants of "gate" like in the previous paragraph.

A. Explicating Some Structural Invariants

We can make the individual disjuncts in $\tilde{R}_{\text{disj}}$ more amenable to the low-level techniques described in the next section by making some local structural invariants explicit in the formula level. In the following we illustrate some typical cases.

It is common in hybrid automaton that some variables never change value in time elapse steps. This applies not only to variables that are untimed by nature (e.g. counters, other discrete data) but also to variables that are used to remember and compute with the values of timed variables observed when executing transitions (e.g. a variable can be used to remember the oil temperature observed when the engine was started). For such a variable x_i, all the flow conditions $Flow_a(l)$ of one component automaton H_a (the one that conceptually owns the variable) are of form "$(\dot{x}_i = 0) \wedge ...$". This allows us to make the time elapse step specific invariant $(x_i' = x_i)$ explicit by rewriting the encoding disjunct $\exists \vec{V}, \delta, \vec{V}' : E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma$ to $\exists \vec{V}, \delta, \vec{V}' : (x_i' = x_i) \wedge E_\delta \wedge C_{\text{Inv}} \wedge C'_{\text{Inv}} \wedge C_\Gamma$. This transformation helps the low-level abstractor to remove quantified variables by applying inlining; on the pure formula level, noticing that $(x_i' = x_i)$ always holds whenever E_δ holds would not be this easy.

A similar case occurs when a variable x_i is used as an "exact clock" whose value changes linearly w.r.t. the elapsed time. As an example, x_1 is such a variable in the automaton "train1" in Fig. 1. In this case, all the flow conditions $Flow_a(l)$ of one component automaton H_a (the one that conceptually owns the clock) are of form "$(\dot{x} = c) \wedge ...$" for some fixed constants c (usually $c = 1$), allowing us to explicate the corresponding conjunct $(x_i' = x_i + c \cdot \delta)$ in the time elapse step encoding disjunct.

Furthermore, if all the σ-synchronizing transitions in one automaton include a common conjunct (e.g. a clock is always reset with $(x' = 0)$), then it can be explicated in the disjunct encoding σ-synchronization. For example, all the "exit1"-synchronizing transitions in the "train1" automaton in Fig. 1 include the conjuncts $(x_1' \leq 0)$ and $(y_1' = 0)$.

IV. Low-level Structural Abstraction

In this section we discuss how the structured representation obtained in previous section can be further simplified and evaluated to construct a Boolean presentation of the abstraction. The low level simplification routines presented in the following can also be used to simplify an unstructured description of a predicate abstraction problem. The idea is to transform the monolithic quantifier elimination problem into a sequence of smaller problems. Compared to the techniques described in previous section, operating on the automaton representation, here the transformation steps work at the level of the syntax of the formula. The simplifications are thus independent of the input formalism, and are possibly able to exploit additional structure of the original model by making cheap syntactic transformations.

For an expression β, let $vars(\beta)$ denote the set of variables occurring in β, and let $\beta[\gamma \leftarrow \alpha]$ denote a copy of β in which every occurrence of the sub-expression γ has been replaced with α.

We work on a Boolean combination of formulas of the form $\exists \vec{U} : \phi$. We assume that there are no nested quantifiers, and all free variables, that is the set $\vec{Q} := vars(\phi) \setminus \vec{U}$, are Boolean. When computing the abstract transition relation (10) of hybrid automata, we have $\vec{U} = \vec{V} \cup \vec{V}' \cup \{\delta\}$ and $\vec{Q} \subseteq \vec{P} \cup \vec{P}'$.

The simplification steps include inlining and syntactic conjunct clustering, and the idea of clustering is further generalized by a technique we call variable sampling.

A. Inlining

From the abstraction equations (10) and (11) we see that the matrix ϕ generally has the form of an n-ary conjunction. We can identify some of the conjuncts that allow simplifying the formula for less expensive quantifier elimination. The following three equivalence-preserving transformations are applied in sequence, until none of them is applicable.

1) $\exists \vec{U} : \beta \wedge (u{=}\alpha) \;\mapsto\; \exists \vec{U} : \beta[u \leftarrow \alpha]$, where $u \in \vec{U}$ and α is an expression such that $u \notin vars(\alpha)$.
2) $\exists \vec{U} : \beta \wedge (q{\Leftrightarrow}\alpha) \;\mapsto\; (q{\Leftrightarrow}\alpha) \wedge \exists \vec{U} : \beta[q \leftarrow \alpha]$, where $q \in \vec{Q}$, and α is a formula with $vars(\alpha) \subseteq \vec{Q} \setminus \{q\}$. When applying this rule, we identify the formulas q and $\neg q$ with $q \Leftrightarrow \mathbf{true}$ and $q \Leftrightarrow \mathbf{false}$ if necessary.
3) $\exists \vec{U} : \beta \wedge (\gamma{\Leftrightarrow}\alpha) \;\mapsto\; \exists \vec{U} : \beta[\gamma \leftarrow \alpha] \wedge (\gamma{\Leftrightarrow}\alpha)$, where γ and α are formulas such that $vars(\gamma) \cap \vec{U} \neq \emptyset$ and $vars(\alpha) \subseteq \vec{Q}$. This only has an effect if γ occurs as a sub-expression in β.

Transformations 1 and 2 each eliminate a variable u or q from the scope of the quantifier. Transformation 3 replaces occurrences of a formula γ with another formula α that does not contain any variables from $\vec{U}$.

As an example, consider the formula $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p_1' \Leftrightarrow (x' > 0)) \wedge (x' = x)$. Using Transformation 1 with $u := x$ and $\alpha := x'$, we inline the frame condition $(x' = x)$ and obtain $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p_1' \Leftrightarrow (x > 0))$. Then, using Transformations 3 and 2 in this order, we get first $\exists x, x' : (p_1 \Leftrightarrow (x > 0)) \wedge (p_1' \Leftrightarrow p_1)$ and then $(p_1' \Leftrightarrow p_1) \wedge \exists x, x' : (p_1 \Leftrightarrow (x > 0))$. As a result, we have obtained a frame condition for p_1 and eliminated p_1' and x' from the scope of the quantifier.

B. Syntactic Conjunct Clustering

From the quantifier elimination problem $\exists \vec{U} : \phi$, where ϕ is an n-ary conjunction, we can identify clusters of conjuncts such that each variable of $\vec{U}$ occurs in at most one of them. Let us rearrange ϕ into conjuncts $\phi_0 \wedge \phi_1 \wedge \cdots \wedge \phi_n$ such that ϕ_0 contains no variables from $\vec{U}$, and for all $1 \leq i < j \leq n$, the sets $vars(\phi_i) \cap \vec{U}$ and $vars(\phi_j) \cap \vec{U}$ are disjoint. Then, $\exists \vec{U} : \phi$ is replaced with the equivalent partitioned formula $\phi_0 \wedge (\exists \vec{U} : \phi_1) \wedge \cdots \wedge (\exists \vec{U} : \phi_n)$. This enables solving a sequence of smaller quantifier elimination problems instead of one large one even when the matrix ϕ is a conjunction.

C. Variable Sampling

We are often able to benefit from a conjunctive partitioning even if the conjuncts are not totally disjoint w.r.t. the set $\vec{U}$. This is particularly evident in the case of time elapse steps, where the δ variable occurs in the same linear atom with many or all data variables, preventing syntactic clustering. We notice, however, that when δ is instantiated to a specific numerical value, clustering wrt. the remaining variables may become possible. In the general setting, assume that $\phi \equiv \phi_1 \wedge \cdots \wedge \phi_n$ and there is a subset $\vec{W} \subseteq \vec{U}$ such that for all $1 \leq i < j \leq n$, the set $vars(\phi_i) \cap vars(\phi_j) \cap \vec{U}$ is a subset of $\vec{W}$. Then, $\exists \vec{U} : \phi$ is equivalent to $\exists \vec{W} : \big((\exists \vec{U} \setminus \vec{W} : \phi_1) \wedge \cdots \wedge (\exists \vec{U} \setminus \vec{W} : \phi_n)\big)$. The sub-problems $\exists \vec{U} \setminus \vec{W} : \phi_i$ cannot be solved directly using efficient SMT-based techniques [6], [7] when $\vec{W}$ contains non-Boolean variables. However, if the values of all variables in $\vec{W}$ are fixed, then the free variables in the sub-problems are all Boolean, and the sub-problems can be solved in sequence. By performing the quantifier elimination when $\vec{W}$ is fixed, we get an under-approximation of $\exists \vec{U} : \phi$. The following procedure shows that by sampling only a finite number of valuations of $\vec{W}$, we can accumulate the precise formula of $\exists \vec{U} : \phi$, which is then returned by the procedure.

1) $\alpha \leftarrow \mathbf{false}$
2) **while** $\phi \wedge \neg\alpha$ is satisfiable:
3) $\quad w \leftarrow$ a valuation of $\vec{W}$ that satisfies $\phi \wedge \neg\alpha$
4) $\quad \alpha \leftarrow \alpha \vee \mathrm{Elim}\big(\neg\alpha \wedge \bigwedge_{1 \leq i \leq n} \exists \vec{U} : \phi_i[\vec{W} \leftarrow w]\big)$
5) **end while**
6) **return** α

On line 3, an SMT solver is used to discover a new valuation of $\vec{W}$. On line 4, the expression $\phi_i[\vec{W} \leftarrow w]$ denotes the assignment of the values w to $\vec{W}$ in ϕ_i, and Elim is a procedure that eliminates the quantifiers from its argument and returns the resulting formula over $\vec{Q}$. The benefit is that Elim can now exploit syntactic conjunct clustering and eliminate the quantifiers in sequence.

The above procedure maintains the invariant that α is an under-approximation of $\exists \vec{U} : \phi$. On the other hand, α is equivalent to $\exists \vec{U} : \phi$ when the formula $\phi \wedge \neg\alpha$ is unsatisfiable, and this is when the procedure returns. Termination of the loop follows from the fact that on line 4, procedure Elim always finds at least one new valuation of $\vec{Q}$ that makes α false and $\exists \vec{U} : \phi$ true, and there is only a finite number of possible valuations of the Boolean variables $\vec{Q}$.

In abstraction problems originating from hybrid systems, the set $\{\delta\}$ is a good candidate for the set $\vec{W}$. Effectively, this means computing the abstraction of the time elapse step in pieces, where each piece corresponds to a fixed value of δ, i.e. a fixed-length time interval.

D. Blocking Visited Models

Once the simplifications have been carried out, quantifier elimination to the remaining parts is applied. We remark that, when computing a disjunction of quantifications, it is possible to restrict the search to the negation of the models computed so far. This can pushed even further by considering that at a certain point of the abstraction computation we may end up with a formula of the form $\alpha \vee (\gamma \wedge \exists \vec{U} : \phi)$, where the subformulas α and γ do not contain quantifiers or variables from $\vec{U}$. A valuation of the free variables $vars(\phi) \setminus \vec{U}$ in $\exists \vec{U} : \phi$ is a don't care if it entails α or $\neg\gamma$. Thus, we can further reduce the models the quantifier elimination has to enumerate by computing the Boolean formula $\exists \vec{U} : (\gamma \wedge \neg\alpha \wedge \phi)$.

V. Related Work

In the Boolean setting, quantification is typically used for the basic operation of Symbolic Model Checking, i.e. image computation. Quantification procedures based on Binary Decision Diagrams (BDDs) have been aggressively optimized [15], [16], [17]. Depending on the nature of the design under verification, the corresponding transition relation was disjunctively or conjunctively decomposed, and quantifiers were pushed inside as to operate on possibly smaller BDDs. The above transformations typically operate on the set of support of BDDs. More recently, SAT-based quantifier elimination has been investigated [18], [19], also in combination with BDD-based techniques [20].

The preliminary idea of pushing quantifiers inside of conjuncts and disjuncts for computing abstraction has been firstly discussed in [21]. However, although the description was done in a general settings, it was applied for abstracting finite domains using BDD operations.

The idea of using decision procedures for computing abstractions has been explored in [22], [23]. The work in [6] improves over them by lifting DPLL-based quantification to the case of SMT. The approach, referred to as All-SMT, is based on the use of an SMT solver, which iteratively finds models that satisfy the formula under the quantifier; each satisfying assignment of the free (Boolean) variables is added as a blocking clause, and the search is restarted until no more models are found. The work in [7] attempts to overcome some of the inefficiencies of [6] by combining BDD-based reasoning and SMT techniques. Compared to the work presented in this paper, both [6] and [7] tackle the problem of predicate abstraction as a monolithic problem. We remark that, any of these two techniques can be used as back-ends in the procedure presented in this paper.

Several approaches trade precision for accuracy. In fact, [6] also shows how to approximate the results. A similar line is followed in [2], [24], 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.

Several approaches to software model checking rely on the construction of an abstract space based on the availability of predicates [25], [26]. In a sense, the approach is exploiting the structure of the control flow graph to partition the problem of computing the abstractions: the abstract system shares the same control flow graph with the program being abstracted. The main difference with our approach is in that here we do not have a single sequential program, but rather a set of

concurrent programs interacting via shared variables and with global timed transitions, which poses additional difficulties.

Several notable abstraction mechanisms have been proposed in settings other than predicate abstraction. The work by Segelken [27] addresses abstraction for concurrent systems by retaining the control structure of the automata being analyzed but dropping information and over-approximating the transition relation. A similar approach is also used in [28] for timed systems, and [29], [30] for hybrid systems.

VI. Implementation and Experiments

We have implemented the proposed techniques within an extended version of the NuSMV model checker [8] that allows for variables of type Real and is connected to the MathSAT SMT engine [31]. This version allows for bounded model checking, and a CEGAR loop based on predicate abstraction. The proposed techniques have been implemented as follows. The networks of automata have been described as data structures in the Python programming language. The high level abstraction is implemented as a Python front-end that generates from the network of automata an abstraction problem as a formula either in monolithic form $\tilde{R}$ or disjunctive form $\tilde{R}_{\text{disj}}$. The lower level computation engine is implemented within NuSMV, and relies on the formula manipulation routines to implement inlining and syntactic conjunct clustering. The quantifier elimination can either be performed by the All-SMT functionalities provided by MathSAT, or by hybrid techniques combining BDDs and SMT [7]. The variable sampling procedure is obtained by integration with MathSAT. In the implementation we exploit the incrementality provided by MathSAT, so that learnt theory lemmas can be reused in the different calls.

For the experimental evaluation, we considered two sets of benchmarks. The first set is taken from the examples distributed together with HyTech [14]. The predicates for the abstraction of this set were found automatically using the CEGAR loop in NuSMV while proving/disproving the property. The second set is obtained by means of a script that generates structured descriptions for networks of linear hybrid automata; the examples contain non-synchronizing and synchronizing transitions and flow constraints, and data variables with a controlled amount of interdependencies. Each predicate for abstraction corresponds to either a location of an automaton being active or to a random linear atom being true. In the experiments we varied the number of parallel automata between 3 and 5, predicates between 13 and 37, data variables between 3 and 39, and top-level disjuncts between 18 and 38. The set contains 756 different models in total.

The problem we consider consists of the generation of the (BDD representing the) transition relation of the system obtained by precise abstraction with respect to the set of predicates. In the experiments we disregard the computation of abstract initial state and property formulas, since they are in general much easier to obtain than the transition relation.

For each of the benchmarks, we compare the proposed abstraction techniques to the direct monolithic quantifier elimination. When moving from monolithic to structural abstraction, we incrementally enable the different techniques to assess their effectiveness individually. In the experiments, we use the MathSAT All-SMT quantifier elimination procedure. The hybrid techniques of [7] shows similar trends. The benchmarks were run on a cluster of Intel Xeon 5130 based machines, with one CPU core allocated for each problem. In all runs, we used a time out of 10 minutes and a memory limit of 1GB. The software and models used in the experimental evaluation are available at http://es.fbk.eu/people/roveri/tests/fmcad09/.

TABLE I
Abstraction results for HyTech models.

Model	$\lvert\vec{P}\rvert$	$\lvert\vec{V}\rvert$	$\lvert$disj$\rvert$	computation time (s) monol.	partit.	clust.	sampl.	sampling clu	sam
active	34	5	27	54.626	18.847	2.410	0.937	5	1
active-trace	34	7	27	51.781	22.171	2.473	0.952	5	1
audio	30	6	15	13.826	4.547	0.448	0.442	2	2
audio-timing	29	7	15	10.910	3.915	0.947	0.690	2	6
billiard-timed	25	3	5	0.910	0.732	0.732	1.044	2	13
dist-controller	8	7	12	0.320	0.232	0.195	0.147	5	1
grc-ver	24	5	11	33.068	19.599	10.421	0.455	4	8
new-grc	22	5	11	38.649	17.840	7.395	0.383	4	7
railroad	16	3	8	0.170	0.140	0.131	0.112	2	5
reactor-clock	19	4	5	0.181	0.133	0.069	0.050	2	2
reactor-rect	17	4	5	0.132	0.112	0.051	0.045	2	2

Table I shows the results for the HyTech automata. The columns $|\vec{P}|$, $|\vec{V}|$, and |disj| show the number of predicates, the number of data variables, and the number of top-level disjuncts in $\tilde{R}_{\text{disj}}$ for each instance. The column "monol." shows the run time in seconds for computing the abstract transition relation using monolithic All-SMT on the compact non-partitioned formula $\tilde{R}$. The column "partit." shows the run time with disjunctive partitioning and inlining enabled. In the column "clust.", syntactic conjunct clustering is enabled in addition, and in the column "sampl.", also the variable sampling of Sec. IV-C is used with $\vec{W} = \{\delta\}$. The listed times are average times over 9 identical runs given a fixed set of predicates. No individual run time was more than 14% off the average. Some HyTech models that were experimented with are omitted from Table I, because in those cases the run time is less than 0.1 s using all four approaches, with no measurable differences.

Fig. 2 shows corresponding run times for the randomly generated hybrid automata. Each marker represents a network of automata and a set of predicates, for which the abstract transition relation was computed once using monolithic All-SMT (run time shown on the x-axis), once using disjunctive partitioning with inlining enabled (the y-axis of Fig. 2(a)), and once with also syntactic conjunct clustering and variable sampling wrt. the δ variable enabled (the y-axis of Fig. 2(b)).

The results clearly show that the presented techniques can dramatically speed up the abstraction computation. Moreover, the enabling of all the features together results in the biggest improvement. We remark that the variable sampling technique is able to increase the number of clusters in all of the HyTech models of Table I (this is not generally always the case), and this often makes the particularly expensive abstraction of the time elapse step much more tractable. The rightmost columns

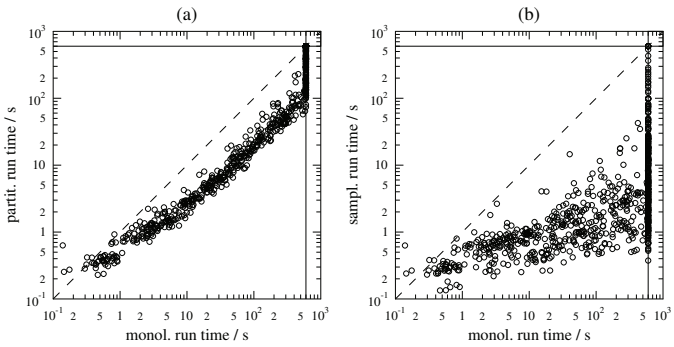


Fig. 2. Abstraction computation times for randomly generated LHA.

"clu" and "sam" show the number of clusters resulting from fixing δ and the number of values of δ that were sampled to obtain the precise abstraction. Variable sampling wrt. variables other than δ might give further speedup, but was not tested in these experiments.

VII. Conclusions and Future Work

In this paper, we have tackled the problem of computing precise predicate abstractions. We have proposed two main contributions. On the high level, we have shown that it is possible to exploit several features common in standard modeling languages in order to obtain high level simplifications. At the low level, we propose to manipulate the logical formulation of the abstraction problem with transformations such as inlining, quantification push-in, and the novel variable sampling. This divide-and-conquer approach results in simpler formulations, that, as shown by the experimental evaluation, can be more effectively computed.

The approach presented in this paper also enables for an easy use of standard optimizations for image computation in the abstract space, e.g. conjunctive and disjunctive partitioning of the transition relation [15], [16], [17].

In the future, we will investigate the application of the proposed techniques for the abstract reachability, according to the lines defined in [22], and a formulation that is able to provide incrementality for the abstraction-refinement loop.

Acknowledgements

The financial support of Helsinki Graduate School in Computer Science and Engineering, Emil Aaltonen Foundation, Academy of Finland (project 112016), and Technology Industries of Finland Centennial Foundation is gratefully acknowledged. A. Cimatti and M. Roveri are sponsored by the European Commission with project FP7-2007-IST-1-217069 COCONUT.

References

[1] E. M. 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.

[2] 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.

[3] M. K. Ganai and A. Gupta, "Completeness in SMT-based BMC for software programs," in *DATE*. IEEE, 2008, pp. 831–836.

[4] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV*, ser. LNCS, vol. 1254. Springer, 1997, pp. 72–83.

[5] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *The Handbook of Satisfiability*. IOS Press, 2009.

[6] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *CAV*, ser. LNCS, vol. 4144. Springer, 2006, pp. 424–437.

[7] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing predicate abstractions by integrating BDDs and SMT solvers," in *FMCAD*. IEEE C. S., 2007, pp. 69–76.

[8] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *STTT*, vol. 2, no. 4, pp. 410–425, 2000.

[9] R. Alur, T. Dang, and F. Ivančić, "Predicate abstraction for reachability analysis of hybrid systems," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 1, pp. 152–199, 2006.

[10] ——, "Counterexample-guided predicate abstraction of hybrid systems," *Theoretical Computer Science*, vol. 354, pp. 250–271, 2006.

[11] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theor. Comp. Sci.*, vol. 138, pp. 3–34, 1995.

[12] R. Alur, T. A. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Trans. on Sw. Eng.*, vol. 22, no. 3, pp. 181–201, 1996.

[13] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "HyTech: a model checker for hybrid systems," *STTT*, vol. 1, pp. 110–122, 1997.

[14] "HyTech: The HYbrid TECHnology tool," May 2009, http://embedded.eecs.berkeley.edu/research/hytech/.

[15] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations," in *VLSI 91*, ser. IFIP Transactions. North-Holland, 1991, pp. 49–58.

[16] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton, "Efficient BDD algorithms for FSM synthesis and verification," in *IEEE/ACM Proc. International Workshop on Logic Synthesis*, May 1995.

[17] D. Geist and I. Beer, "Efficient model checking by automated ordering of transition relation partitions," in *CAV*, ser. LNCS, no. 818. Springer, 1994, pp. 299–310.

[18] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 250–264.

[19] 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.

[20] O. Grumberg, A. Schuster, and A. Yagdar, "Hybrid BDD and All-SAT Method for Model Checking," in *Symposium on Satisfiability Solvers and Program Verification (SSPV)*, Seattle, USA, Aug. 2006.

[21] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[22] S. K. Lahiri, R. E. Bryant, and B. Cook, "A symbolic approach to predicate abstraction," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 141–153.

[23] S. K. Lahiri, T. Ball, and B. Cook, "Predicate abstraction via symbolic decision procedures," in *CAV*, ser. LNCS, vol. 3576. Springer, 2005, pp. 24–38.

[24] D. Kroening and N. Sharygina, "Image computation and predicate refinement for RTL Verilog using word level proofs," in *DATE*. ACM, 2007, pp. 1325–1330.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[26] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 570–574.

[27] M. Segelken, "Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models," in *CAV 2007*, ser. LNCS, vol. 4590. Springer, 2007, pp. 433–448.

[28] S. Kemper and A. Platzer, "Sat-based abstraction refinement for real-time systems," *ENTCS*, vol. 182, pp. 107–122, 2007.

[29] A. Tiwari, "Abstractions for hybrid systems," *Formal Methods in System Design*, vol. 32, no. 1, pp. 57–83, 2008.

[30] M. Fränzle, "Verification of hybrid systems," in *CAV*, ser. LNCS, vol. 4590. Springer, 2007, p. 38.

[31] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MathSAT 4SMT Solver," in *CAV*, ser. LNCS, A. Gupta and S. Malik, Eds., vol. 5123. Springer, 2008, pp. 299–303.

Enhanced Verification by Temporal Decomposition

Michael L. Case Hari Mony Jason Baumgartner Robert Kanzelman
IBM Systems and Technology Group

***Abstract*— This paper addresses the presence of logic which has relevance only during initial time frames in a hardware design. We examine transient logic in the form of signals which settle to deterministic constants after some prefix number of time frames, as well as primary inputs used to enumerate complex initial states which thereafter become irrelevant. Experience shows that a large percentage of hardware designs (industrial and benchmarks) have such logic, and this creates overhead in the overall verification process. In this paper, we present automated techniques to detect and eliminate such irrelevant logic, enabling verification efficiencies in terms of greater logic reductions, deeper Bounded Model Checking (BMC), and enhanced proof capability using induction and interpolation.**

I. Introduction

Automated verification of sequential hardware designs is often a computationally challenging task. Being a PSPACE problem, the size of the design under verification can often render an automated solution intractable. Many hardware designs contain extraneous artifacts which are largely irrelevant to the core verification task, yet whose presence creates bottlenecks to the verification process. This paper addresses two particular types of artifacts: *transient signals* which after a certain number of time steps settle to a fixed constant value, and *initialization inputs* used to encode intricate initial states which become irrelevant after a certain number of time steps. We present algorithms to automate the identification of such artifacts, as well as to eliminate these artifacts to enhance the overall verification process.

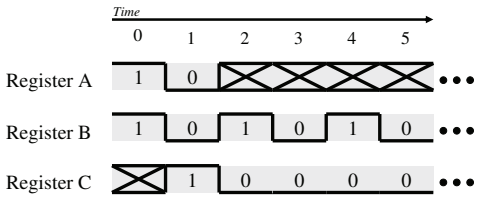


Fig. 1. Register C is a transient signal. X-valuations designate undetermined valuations, i.e. the possibility of either a 0 or 1 value.

Definition 1 (Transient Signal): Let $x(D,t)$ denote the value of signal x in design D at time t. A transient signal is any Boolean signal s such that $\exists T \in \mathbb{N}.\ \exists C \in \{0,1\}.\ \forall$ time $t \geq T.\ s(D,t) \equiv C$. The smallest T for which this holds is referred to as the *transient duration* and C as the *settling constant*.

In Figure 1, register C is a transient because it takes value 0 starting at time 2 and holds this value for all time. Registers A and B are not transients because they remain undetermined in value and do not settle to a fixed constant value, respectively. We have found that transient signals occur frequently both in industrial and benchmark designs.

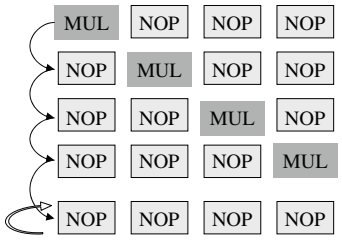


Fig. 2. FPU pipeline as driven by the verification environment

Transient logic arises due to numerous causes. One such cause is the presence of an initialization sequence. A common design style allows a design to power-up in a non-deterministic state from which sequence of state transitions bring it to a "reset state" from which it behaves consistently. Commonly dedicated logic is used to force the design through its initialization phase, and verification assumptions/checkers are tailored to take effect only once the initialization phase is completed. Much of the logic settles to a fixed constant value once the initialization is complete, and this logic is amenable to simplification by our proposed techniques.

Numerous methodologies have been proposed to eliminate such overhead, – e.g. by using three-valued simulations of the initialization phase (applying X-values to reflect non-determinism, as depicted in Figure 1) to conservatively determine a set of states the design can reside in post-initialization [1], [2]. While such methodologies are well-motivated, they require dedicated manual effort to decompose the overall verification task. They also ultimately over-approximate the set of post-initialization states since they conservatively treat non-deterministic signals as non-constant, losing more subtle constraints which may exist over the post-initialization states. This latter problem in turn may prompt a logic designer to conservatively add to the initialization logic to avoid spurious failures, a manual process which may result in suboptimal silicon.

Transient logic may also arise due to the verification testbench itself. Testbenches consist of three components: 1) a *driver* containing enough input assumptions to provide meaningful input stimulus, 2) the design under verification, and 3) a *checker* to verify the correctness of the design under the given inputs. The driver may be constructed to over-constrain the inputs of the design, e.g., to test the design only against a subset of its possible behaviors to facilitate a case-splitting strategy. Under this reduced set of inputs, many internal design signals may settle to constant behavior after a certain number of time steps.

An example of transient logic arising from the verification testbench can be found in the Floating-Point Unit (FPU) verification methodology proposed in [3]. To help cope with

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

the complexity of such verification, this methodology checks the correctness of a single opcode propagating through an empty pipeline as depicted in in Figure 2. No-operation (NOP) opcodes are driven after the single opcode under evaluation, and after the meaningful opcode is evaluated the internal state of the FPU settles to constant NOP behavior. All signals in the FPU may thus be viewed as transient logic when the inputs are driven in this manner.

A related problem is that of extraneous initialization inputs.

Definition 2 (Initialization Input): Let Σ be a set containing the design's next state functions, properties, and constraints. An *initialization input* is a primary input whose value can only propagate to any $\sigma \in \Sigma$ at time 0. At all other times, the value of the initialization input is not observable with respect to Σ.

It is common for designs to have a set of multiple possible initial states. The driver can non-deterministically select a single initial state from this set by introducing primary inputs. These are initialization inputs because their values after the first time frame are irrelevant. The method of eliminating transient signals presented in this paper produces designs with large sets of possible initial states and many initialization inputs. Therefore we propose a method to identify a subset of these initialization inputs that can be safely replaced with constant values, enhancing our ability to eliminate transients from designs without significantly increasing the total size of the design.

The contributions of this paper are as follows:

1) We propose an algorithm to identify the existence and duration of transient signals in Section II-A. This method utilizes ternary simulation hence is very fast and scalable.

2) We propose an efficient method to eliminate transient logic in Section II-B. This works by decomposing an unbounded verification problem into two sub-problems: bounded verification over the initial time frames during which transient behavior occurs, and unbounded verification over the remaining time frames. The unbounded verification can safely assume that all transients have settled to their post-transient constant values and thus can simplify the complex unbounded verification problem.

3) We propose a technique to identify initialization inputs that can safely be replaced by constants in Section III. This method uses bounded model checking combined with structural analysis, and thus is scalable. It is useful to eliminate inputs that exist in the testbench to model power-on non-determinism. Our method of eliminating transient logic can manufacture such inputs, so as well as being a generally useful simplification, the technique to eliminate initialization inputs in Section III is particularly useful after transient logic is eliminated.

4) We provide a large set of experimental results throughout this paper to illustrate the benefits of our techniques across numerous academic and industrial designs.

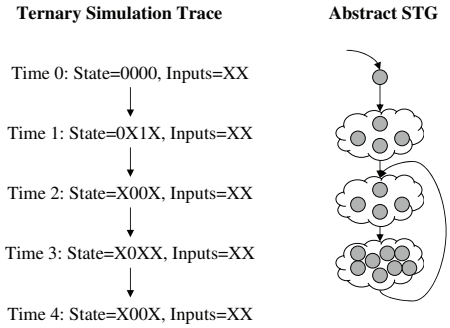


Fig. 3. Example ternary simulation execution

II. Temporal Decomposition

A. Detection of Transient Signals

Ternary simulation is a method to efficiently over-approximate the set of reachable states in a design. It works by conservatively modeling primary inputs with ternary X values and simulating a sequence of 3-valued states until a state is repeated. Upon convergence, the set of observed 3-valued states constitutes an over-approximation to the set of reachable states. An example ternary simulation run is depicted in Figure 3, where the state at time 2 is repeated at time 4 indicating that the reachable states have been over-approximated.

While the over-approximation of such ternary simulation is often too coarse for property checking itself, it is useful to efficiently identify useful design characteristics. For example, certain constant and equivalent signals may be detectable using this approach, and this enables the design to be simplified. Clock-like oscillating signals may also be detectable which enable temporal phase abstraction [5].

Ternary simulation can be augmented to efficiently detect a subset of the transient signals, in addition to the transient duration after which they settle to constant behavior.

Theorem 1 (Transient Detection by Ternary Simulation): Let $x(S,t)$ denote the value of signal x in the 3-valued state seen at time t. Let $i < j$ such that $\forall x$, $x(S,i) = x(S,j)$. If $\exists\, C \in [0,1]$, $\exists\, \sigma$ such that $\forall\, k \in [i,j]$, $\sigma(S,k) = C$ then σ is a transient signal with transient duration at most i.

It is straight-forward to use ternary simulation as per Theorem 1 to find transient signals, and a simple implementation[1] is shown in Algorithm 1. After convergence, a sweep over all signals is performed to see which remained constant within the state-repetition loop. Those which remained constant are added to the transient list, along with the constant they settled to and the latest time frame at which they evaluated the non-constant value. This represents an upper-bound (due to the over-approximation of ternary evaluation) on their transient duration.

B. Simplification of Transient Signals

Given a set of transient signals, consider the maximum transient duration within this set. Before the maximum duration,

[1]It is possible to optimize Algorithm 1 by incorporating the transient signal book-keeping within the ternary simulation loop.

Algorithm 1 Detection of Transient Signals

```
1: function detectTransients(design)
2:    // Execute ternary simulation until convergence
3:    History := ∅
4:    ternaryState := getTernaryInitialState(design)
5:    for (time = 0; ; time++) do
6:       if (ternaryState ∈ History) then
7:          cycleStartTime := calculateCycleStartTime(History, ternaryState)
8:          break
9:       end if
10:      History := History ⋃ ternaryState
11:      ternaryState := getNextTernaryState(design, ternaryState)
12:   end for
13:
14:   // Extract the transient signals
15:   transients := ∅
16:   for all (signals s in design) do
17:      for all (constants C in {0, 1}) do
18:         if (∀ time > cycleStartTime, s = C) then
19:            duration := latestToggle(s, History)   // ≤ cycleStartTime
20:            transients := transients ⋃ (s, C, duration)
21:         end if
22:      end for
23:   end for
24:
25:   return transients
26: end function
```

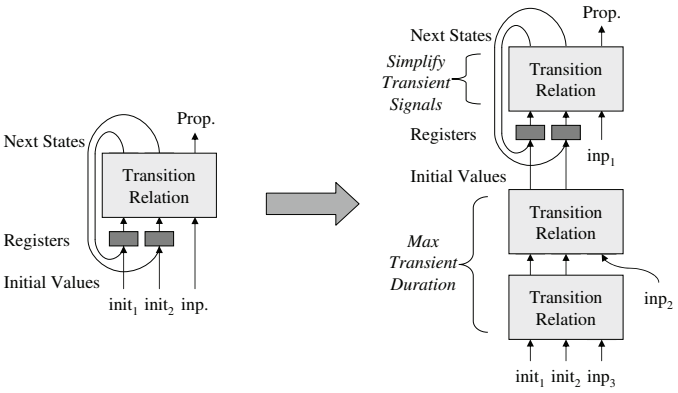


Fig. 4. Time-Shifting a design and simplifying transient signals

one or more of these signals may assume values which differ from their settled constants. We use BMC [6] to check the validity of any properties on these initial time frames. After the maximum duration, all transients in the set have settled to their corresponding constant values. The design can be simplified by replacing these transient signals with their constant values, and an unbounded verification process can check the remainder of the time frames in the simplified model.

We wish to optimize unbounded-verification within a transformation-based verification (TBV) framework [4]. In such a framework, a sequence of transformations is applied to a design (often represented as a *netlist*) prior to the invocation of a terminal verification algorithm, allowing the reductions inherent in the former to yield substantial speedups to the latter. As such, we cast our transient simplification routine as a netlist transformation rather than a customization to an existing verification algorithm. This allows these simplifications to be compatible with any downstream synthesis or verification algorithm.

In order to check properties only after the transient duration using a general-purpose verification algorithm, it is necessary to *time-shift* the design. In this process, we adjust the time basis of the design such that time *maxTransientDuration* in the original design corresponds to time 0 in the time-shifted design. Figure 4 demonstrates how such shifting may be accomplished. The design's initial state is modified such that the time-shifted design starts in any state reachable in *maxTransientDuration* time steps. In this work, we achieve such a transformation by unrolling the transition relation and driving the initial states with the output of the unrolled structure. This is akin to using structural symbolic simulation to compute the new set of initial values.

Algorithm 2 illustrates a simple procedure that will use a set of detected transient signals to simplify the design. BMC is used to check the properties before the maximum transient duration. Then the design is time-shifted, and the netlist is simplified by merging transients with their respective settling constants. We limit the runtime to 10-seconds because most of the benefits are obtained quickly at low time t.

Algorithm 2 Simplification of Transient Signals

```
1: function simplifyTransients(design, transients)
2:    maxTransientDuration := computeMaxTransientDuration(transients)
3:    for (t = 0; (t < maxTransientDuration); t++) do
4:       Validate that properties are not falsifiable at time t with BMC
5:
6:       // Time shift by 1 clock cycle
7:       for all (registers r ∈ design) do
8:          initialValue(r) := nextState(r)
9:       end for
10:
11:      // Incrementally simplify the design
12:      for all (g ∈ transients) do
13:         if (transients[g].settlingTime ≤ t) then
14:            merge(design, g, transients[g].settledConstant)
15:         end if
16:      end for
17:   end for
18: end function
```

C. Results

All experiments in this paper are performed on a set of 135 difficult industrial verification problems seen within IBM and 28 difficult HWMCC '08 designs [7]. The IBM benchmarks are from a suite of microprocessor property checking and sequential equivalence checking benchmarks that are not easily solved with BMC, interpolation, or induction.[2] The HWMCC benchmarks were either unsolved in HWMCC '08 or only solved by a few of the entrants. The size of the sequential designs varied with the largest design consisting of 97,000 registers and 632,000 and gates (when expressed as an AIG [12]).

Table I examines the transients that were present in some of these benchmarks. The columns "T. Found" show the number of transients found with Algorithm 1. In total, transients were present in 48.6% of the industrial designs and 25.0% of the HWMCC designs. On average, there were 22.7 transient signals in each industrial design that had transients and 4 transients in each corresponding HWMCC design.

[2]Each algorithm was run for 10 minutes. Induction incrementally increases k until the 10 minute timeout is reached (or a property is verified).

TABLE I. Transients Found and Simplified, On a Subset of Our 135 IBM and 28 HWMCC Designs

	Design Size			T. Found		Transients Simplified		T. Found With Induction		Logic Size Change	
Design	ANDs	Regs	Inputs	Num.	Dur.	Num.	Dur.	Num.	Runtime	TR	Total
IBM0	3039	291	45	8	1	8	1	120	4.18 s	-1.10%	1.83%
IBM6	77313	9829	544	1	1	1	1	0	0.16 s	-1.44%	2.05%
IBM8	3011	396	86	2	2	2	2	12	11.50 s	-0.27%	2.76%
IBM22	11218	908	924	23	10	23	10	36	9.48 s	-17.17%	19.48%
IBM23	8713	477	6	1	1	1	1	0	0.17 s	-9.57%	0.00%
IBM28	116322	21607	1316	1150	74	0	0	0	64.09 s	0.00%	0.00%
IBM31	7291	696	61	4	3	4	3	0	44.08 s	-2.23%	7.26%
IBM32	8177	698	65	4	3	4	3	7	49.36 s	-1.18%	7.18%
nusmvbrp	464	52	11	2	1	2	1	5	0.07 s	-8.18%	3.18%
nusmvguidancep2	1748	86	84	2	1	2	1	0	0.39 s	-1.89%	15.14%
nusmvqueue	2376	84	82	2	1	2	1	0	0.28 s	-4.10%	11.79%
nusmvreactorp2	1242	76	74	2	1	2	1	0	0.35 s	-5.52%	15.82%
r17b	4087	322	613	18	2	16	1	1	3.30 s	-30.38%	44.03%

Our implementation of Algorithm 2 bounds the total runtime at 10 seconds, and hence not all designs that had transients were simplified. Columns "Transients Simplified" show the transients that were utilized for simplification: in total 43.0% of the IBM designs and all 25.0% of the HWMCC designs were simplified. The transient simplification process entailed time-shifting the industrial designs by an average of 4.5 time frames, and the HWMCC designs were each time-shifted by 1 time frame.

Algorithm 1 uses ternary simulation to find transients. This procedure is fast – merely finding transients takes less than 1 second on all of our benchmarks – but lossy. As an alternative, consider time-shifting the design by 6 time frames, sufficient to capture 83% of the transients found with ternary simulation, and then using an inductive[3] routine to find sequentially-constant signals in the modified design. Each constant signal is a transient, assuming the constant did not exist before time-shifting. The columns "T. Found With Induction" illustrate the results of this experiment. On 24.1% of the designs our ternary simulation procedure found more transients than the inductive procedure, indicating that many of the found transients are non-inductive. On 53.8% of the designs the inductive procedure found more, indicating that on these designs induction was more complete than ternary simulation. The ternary simulation procedure was often orders-of-magnitude faster than the inductive procedure, and for this reason Algorithm 1 uses ternary simulation.

Columns "Logic Size Change" show the effect of transient simplification on the logic size (after finding transients with ternary simulation). Transient simplification decreased[4] the size of the transition relation (TR) in the IBM designs by 1.3% and the transition relations of the HWMCC designs by 7.4%. However, when combined with time-shifting, the resultant designs had complex initial value logic which led to an increase in size by 16.1% on the IBM designs and 25.8% on the HWMCC designs. This increase can be mitigated by removing unnecessary initialization inputs, as described in Section III.

III. Initialization Inputs

As discussed in Section I, some testbenches contain initialization inputs to model complex initial values. Additionally, initialization inputs often arise due to the symbolic simulation used in time-shifting netlists for the simplification of transient signals. (A similar complexity of initial values occurs as a byproduct of peripheral retiming [4], [8].) The increase in size due to the time-shifted initial values is undesirable, in that it may offset the reduction in size resulting from the merging of transient signals to constants. While certain algorithms such as induction may be immune to increases in initial value complexity, in a TBV setting some algorithms may be hindered by this increased complexity. We thus are motivated to simplify complex initialization logic.

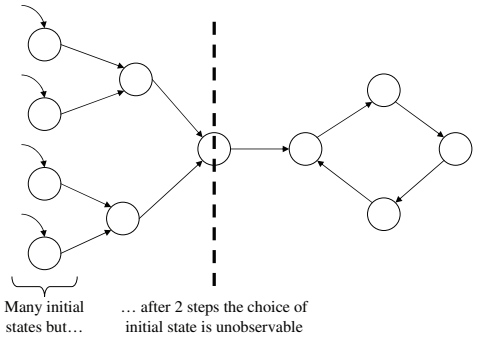


Fig. 5. STG illustrating the irrelevance of certain initialization inputs

A. The Overhead of Unnecessary Initialization Inputs

Recall that an initialization input is one whose value only affects the design at time 0, often used to encode complex initial states (Definition 2). Time-shifting inherently introduces a significant number of initialization inputs (eg.: Figure 4). Not all of these inputs may be relevant to the behavior of the design, and the time-shifted design can be optimized by removing a subset of the new inputs.

Consider the state-transition graph (STG) shown in Figure 5. This design can start in four possible initial states, and hence two initialization inputs may be used to model this set of initial states. All paths in the STG pass through a single dominator state after two time steps, and so for time $t \geq 2$ it is possible to reduce the number of possible initial states without affecting the design's behavior. As the set of initial values is represented in a netlist using extra initialization inputs, such a simplification may be performed by replacing these initialization inputs by constants. This type of simplification is therefore a form of Observability Don't Care (ODC)-based simplification [9] because the individual initial states are not

[3] $K = 5$ induction with unique-state constraints

[4] The size change is measured as $1 - \frac{\text{new regs.}+\text{new ands}+\text{new inputs}}{\text{old regs.}+\text{old ands}+\text{old inputs}}$.

observable after a finite number of time steps.

B. Reduction of Initialization Inputs

A subset of initialization inputs which are irrelevant to the behavior of the design may be detected using structural analysis alone. Theorem 2 illustrates how unrolled Cone of Influence (COI) analysis can be used to identify a subset of inputs that have no influence on the netlist after a fixed number of time steps t.

Theorem 2 (Detection of Irrelevant Initialization Inputs): Recall that $x(D,t)$ denotes the value of signal x in design D at time t. Let UD be a temporal unrolling of D and let Σ be the set of all next-state functions and properties in D. If $\exists\, t \in \mathbb{N}$ and input i s.t. $\forall\, \sigma \in \Sigma, i \notin COI(\sigma(UD,t))$ then we can replace i with an arbitrary signal to produce design D' s.t. $\forall\, t > T$, $\forall\, \sigma \in \Sigma$, $\sigma(D',t) = \sigma(D,t)$.

Once an irrelevant initialization input is identified, it may be replaced with an arbitrary value[5] without affecting the behavior of the design after time t. The modified design is guaranteed to be equivalent to the original design after time t, but before time t the modified design can only visit a subset of the states of the original design. To ensure that valid counterexamples are not missed during this simplification, it is necessary to validate the correctness of the properties up to time t before this simplification. (This is similar to the initial property validation necessary before merging transient signals in the time-shifted netlist as per Algorithm 2.)

Algorithm 3 Eliminate Irrelevant Inputs

```
1: function simplifyInputs(design, maxTime)
2:   unrolledModel := ∅
3:   for all (registers r ∈ design) do
4:     instance of r in unrolledModel := initialValue(design, r)
5:   end for
6:   for (t = 0; t < maxTime; t++) do
7:     Validate that properties are not falsifiable at time t
8:
9:     // Incrementally unroll the model
10:    Append one transition relation to unrolledModel
11:    unrolledModel := resynthesize(unrolledModel)
12:
13:    // Compute the unrolled COI
14:    C := ∅
15:    for all gates g ∈ ( next-state functions ⋃ properties ) do
16:      C := C ⋃ COI(unrolledModel, last temporal instance of g)
17:    end for
18:
19:    // Remove unnecessary inputs
20:    for all (primary inputs i ∈ design) do
21:      if (i ∉ C) then
22:        merge(design, i, 0)
23:      end if
24:    end for
25:  end for
26: end function
```

Algorithm 3 illustrates our initialization input elimination routine. Time t is gradually increased[6] until computational resources are exceeded. For each t, we first validate that the properties cannot be falsified at that time frame. Next, the design is incrementally unrolled and its COI is inspected. In order to reduce the size of this COI and enhance the reduction potential of this technique, synthesis algorithms (SAT-sweeping [10], rewriting [11], and others) are employed on the unrolled design. Inputs that fall out of the COI of the next-state functions and properties are removed from the design by merging them with the constant 0.

We note that unlike most ODC-based simplification routines, all simplifications identified by this routine are inherently compatible. The simplifications can be utilized simultaneously without interfering with one another, resulting in a more efficient algorithm. Additionally, because the approach relies on the circuit structure it is highly scalable. Nonetheless, this technique is incomplete and some irrelevant initialization inputs may not be identified. It can be complemented by a post-processing of traditional, yet more expensive, ODC-based simplification (Section III-C.2).

[5]Our implementation replaces such unnecessary inputs with the constant 0.

[6]We limit $t < 5$ because the simplification potential quickly saturates as t increases. 91.0% of all simplifications are done with $t = 0$, 5.7% with $t = 1$, and $< 1.5\%$ with each of $t = 2, 3, 4$.

TABLE II. Input Simplification, On A Subset of Our Designs

	Design Size			Input Simplification	
Design	ANDs	Regs	Inputs	Init. Inputs	Removed
IBM0	3039	291	45	0	0
IBM6	77313	9829	544	0	0
IBM8	3011	396	86	1	0
IBM22	11218	908	924	275	201
IBM23	8713	477	6	0	0
IBM28	116322	21607	1316	232	13
IBM31	7291	696	61	0	0
IBM32	8177	698	65	0	0

C. Results

1) Initialization Input Simplification: Our first set of experiments considers the impact of our irrelevant input elimination technique alone. A subset of these results are provided in Table II. This simplification technique was effective on 22.4% of the industrial designs, and within those designs 11.7% of the initialization inputs were removed.

We were unable to find any initialization inputs in the HWMCC designs.

2) Transient Simplification + Initialization Input Simplification: Our next set of experiments considers the ability of the irrelevant input elimination technique to reduce the overhead of the symbolic simulation necessary to compute the initial values of the time-shifted netlist produced in transient simplification. Our implementation first eliminates any identified transient signals, and then attempts to eliminate the resulting initialization inputs. A selection of our results is shown in Table III. The transient simplifications are illustrated for each design, and the number of initialization inputs after this simplification step are counted. Initialization input elimination was able to simplify 58.7% of the industrial and 14.3% of the HWMCC designs. The logic bloat of these combined simplification steps was 11.7% on the industrial designs and 9.4% on the HWMCC designs (compare to 16.1% and 25.8%, respectively, without initialization simplifications). This demonstrates that initialization input simplification is effective in mitigating the bloat that is caused by transient simplification.

TABLE III. Input Simplification After Transient Simplification, On a Subset of Our 135 IBM and 28 HWMCC Designs

	Design Size			Transient Simp.		COI-Based Input Simp., Alg. 3			Total	ODC-Based Input Simp.	
Design	ANDs	Regs	Inputs	Num.	Dur	Init. Inputs	Removed	Time	Bloat	Inp. Removed	Time
IBM0	3039	291	45	8	1	25	0	1.59 s	0.73%	0	4.82 s
IBM6	77313	9829	544	1	1	409	1	9.55 s	1.77%	92	3610.39 s
IBM8	3011	396	86	2	2	58	12	0.67 s	2.18%	0	13.78 s
IBM22	11218	908	924	23	10	368	214	1.61 s	-12.35%	0	26.20 s
IBM23	8713	477	6	1	1	0	0	0.00 s	-9.57%	0	0.00 s
IBM28	116322	21607	1316	0	0	0	4	0.34 s	10.49%	4	109.29 s
IBM31	7291	696	61	4	3	178	3	0.36 s	4.99%	0	36.58 s
IBM32	8177	698	65	4	3	178	3	0.37 s	5.96%	0	34.81 s
nusmvbrp	464	52	11	2	1	11	0	0.30 s	-5.00%	0	0.49 s
nusmvguidancep2	1748	86	84	2	1	84	0	1.42 s	13.25%	33	13.09 s
nusmvqueue	2376	84	82	2	1	82	0	1.86 s	7.69%	0	23.05 s
nusmvreactorp2	1242	76	74	2	1	74	0	1.57 s	10.30%	1	18.00 s
r17b	4087	322	613	16	1	558	3	9.25 s	13.54%	17	698.12 s

Algorithm 3 uses BMC and structural analysis to remove initialization inputs, and because of this it is fast yet does not identify all unnecessary initialization inputs. In columns "ODC-Based Input Simp." of Table III we illustrate the relative utility of Algorithm 3 by running a more thorough initialization-input simplification method afterward. This technique is similar to traditional ODC-based optimization algorithms in that it creates a side copy of a logic window of the design for each candidate simplification, assessing whether a particular simplification may be witnessed as altering design behavior with respect to that window [17]. This particular technique was limited to assessing the validity of merging initialization inputs relative to a logic window of a configurable sequential depth. These results illustrate that our method identifies 44.5% of the unneeded initialization inputs in just a fraction of the time needed to do the more exhaustive check; the former was limited to a runtime of 10 seconds total (identical to transient simplification), whereas the latter was limited to 10 seconds per initialization input – resulting in runtimes in excess of 1 hour in numerous cases.

3) Runtime: Most results presented above were presented without runtimes. Transient simplification primarily leverages ternary simulation and BMC, and initialization simplification leverages structural methods and BMC. These techniques are efficient and scalable, and most analysis can be performed incrementally. We thus have designed both of these simplification techniques to incrementally simplify the design over time frames until some computational limit is exhausted. All experiments were run on a cluster of 2 GHz IBM POWER CPUs, and the combined runtime for both transient simplification and initialization simplification was limited to 20 seconds. We often omit runtimes in our tables because this 20-second overhead is negligible in the verification of these complex designs.

IV. Discussion and Experimental Results

The algorithms described above were implemented in the IBM internal verification tool *SixthSense* [8]. *SixthSense* is built upon a TBV framework where various engines incrementally simplify complex properties before leveraging a terminal verification engine to attempt to solve the simplified problem. The techniques presented in this paper have been implemented as one of these engines. In this section we analyze the impact of these techniques on other synthesis and verification algorithms.

TABLE IV. BMC Depth Comparison

Benchmark Set	30-min BMC	Input Simp. + 29-min BMC	Trans. Simp. + Input Simp. + 29-min BMC
IBM (135)	100.00%	100.00%	102.20%
HWMCC (28)	100.00%	100.00%	103.68%
total (163)	100.00%	100.00%	102.39%

A. Relationship with Bounded Model Checking

First we focus on BMC. We compare three different flows:

1) 30-minute BMC
2) Initialization input simplification + 29-minute BMC
3) Transient simplification + initialization input simplification + 29-minute BMC

Note that the combination of transient simplification and initialization input simplification is restricted to run for no more than 20 seconds. This means that the total runtime for each of the three configurations should be approximately the same, with a slight advantage given to Flow 1.

Table IV compares the depths achieved by BMC before the timeout was reached. We normalize all depths to Flow 1 and observe that initialization input simplification alone was not effective in increasing the attained BMC depth. However, transient simplification was effective in increasing the depth by 2.4%. This indicates that BMC was wasting runtime analyzing the signals that were transient, and by removing these signals we enable BMC to do more work in the same amount of time.

B. Relationship with Interpolation

Interpolation [13] is an unbounded SAT-based proof technique. While effective on numerous complex designs, interpolation alone was insufficient to solve many of the benchmarks under analysis.

Table V illustrates the effect of running transient simplification and initialization input simplification before interpolation. Of the industrial designs, 10 designs (13.5%) which interpolation initially could not solve within a 30-minute timeout become solvable after the design is simplified using our proposed techniques. Unfortunately, there were also 4 designs (3.0%) that were previously solvable but are not

TABLE V. Interpolation Comparison

	Transient + Input Simp.	
Benchmark Set	Enables Proof	Breaks Proof
IBM (135)	10	3
HWMMC (28)	0	1
total (163)	10	4

TABLE VI. Design Size after Signal Correspondence, On 163 Designs

Flow	Ands	Regs.
Baseline (after Comb. Synth.)	100.00%	100.00%
Signal Correspondence	76.23%	73.91%
Input Simp. + Signal Correspondence	75.03%	73.89%
Trans. Simp. + Input Simp. + Signal Correspondence	75.39%	72.21%

TABLE VII. Design Size after Retiming, On 163 Designs

Flow	Ands	Regs.	Inputs
Baseline (after Comb. Synth.)	100.00%	100.00%	100.00%
Retiming + Retiming	116.81%	91.06%	174.56%
Algorithms 2,3 + Retiming	118.51%	90.85%	177.55%
Retiming + Algorithms 2,3	108.04%	90.73%	163.65%

solvable after the design is simplified. This illustrates the inevitable instability of the underlying interpolation algorithm, yet we nonetheless are encouraged that our simplification helps interpolation substantially more often than it hurts. In a robust verification setting, it may be desirable to invoke interpolation both before and after our simplifications to maximize the chances of obtaining a conclusive result.

C. Relationship with Signal Correspondence

The detection and simplification of sequentially equivalent signals, sometimes referred to as *signal correspondence* is an effective way to reduce the size of a sequential design [14]. Often these reductions either prove the safety properties or are effective in simplifying the problem for another downstream verification engine.

Table VI shows how our proposed algorithms interact with signal correspondence on the combined set of IBM and HWMCC designs. Combinational synthesis is run on all benchmark designs, and all other design sizes are normalized to this baseline. Signal correspondence is able to reduce the number of Ands in an AIG representation of the design by 23.7% and the number of Registers by 26.1%. Running initialization input simplification prior to signal correspondence does not noticeably help the results, but running transient simplification followed by input simplification prior to signal correspondence is effective.

Using our simplification algorithms before signal correspondence enables signal correspondence to achieve 0.9% better And count and 1.7% better Register count. Note that our simplifications can at times increase the size of the design (due to an increase in the initialization logic, Section II-C). Despite this bloat, signal correspondence is able to achieve a smaller design size than it otherwise would without our simplifications.

D. Relationship with Retiming

Our proposed techniques are related to retiming [4] in that both approaches time-shift the design and, as a byproduct, entail complicated initial values. Table VII explores interleavings of these two techniques on the combined set of IBM and HWMCC designs.

The size of the designs after combinational synthesis is used as a baseline for comparison. Running min-area retiming twice in succession decreases the number of registers but increases all other size metrics, on average.

Running transient simplification and initialization input simplification instead of the first retiming pass slightly decreases the number of registers, but also increases the logic bloat because the initial values are being complicated by both techniques.

The best configuration is where the second retiming pass is replaced with transient simplification and initialization input simplification. This decreases the size of the design across all metrics: number of AIG Ands, number of Registers, and number of Inputs. There are two explanations for this:

- Retiming creates initialization inputs which our techniques are able to eliminate. To test this, we ran just initialization input simplification after retiming and observed our simplification technique reducing Ands by 0.3%, Registers by 0.1% and Inputs by 4.5%. This accounts for some of the reductions of Table VII.
- Transient signals remain in the post-retiming design which Algorithm 2 is able to eliminate.

E. Relationship with Sequential Equivalence Checking and Induction

We have found numerous cases where time-shifting and simplifying transient signals has been crucial to completing a proof by induction, and therefore our techniques are a vital part of the overall scalability of various verification methodologies. Particularly, in sequential equivalence checking (SEC), identification of internal points that are pairwise equivalent is critical to the successful completion of an inductive proof of input-output equivalence. Many of these pairs of internal points are initially inequivalent due to power-on non-determinism, and the inductive proof of input-output equivalence fails because the internal equivalences no longer hold. Numerous techniques have been proposed to decouple the verification of the reset mechanism from post-reset SEC, after which power-on inequivalence has been normalized out [2], but there exist many verification problems that do not benefit from these approaches.

In one particular industrial SEC problem with 804 Registers, 21,726 And gates, and 175 Inputs, the use of time-shifting to eliminate transient signals of duration 7 (due to an "initializing" state machine) followed by the subsequent enhanced synthesis optimization rendered a $k = 2$-inductive sequential equivalence check with a combined overall runtime of 42 seconds. Without the use of our transient-elimination algorithms, induction could not solve the problem even within a runtime of 2 hours and depth $k = 12$.

In another industrial SEC example with 3201 Registers, 19,506 And gates, and 843 Inputs, a 6-frame time-shift to eliminate an "initializing" state machine transients enabled a $k = 1$-inductive sequential equivalence check using a combined runtime of 56 seconds. Without the elimination of transients, the problem was not k-inductive even with depth 5; our only solution to this problem relied upon the use of substantially heavier-weight algorithms such as abstraction and interpolation, and required manual tuning to solve.

As discussed in Section IV-D, transient elimination is related to peripheral retiming in that both techniques can time-shift the design. However, we have found transient elimination

to be more desirable than retiming in certain facets of SEC flows in that the scalability of SEC relies to some extent upon name- and structure-based correlation of registers across the designs being equivalence-checked. Retiming may arbitrarily alter register placement, diminishing the usability of such SEC heuristics.

V. Related Work

Techniques presented in this paper are similar to Kth invariants [15]. In the previous work, equivalences that hold sequentially in all time frames after time K are found by induction and leveraged in unbounded verification of safety properties. This work differs from our own in the following ways:

- In [15], leveraging Kth-invariants requires a specialized verification algorithm that performs backward reachability. Our technique transforms the design such that any downstream verification algorithm in our TBV flow can leverage the information.
- The previous work assumed that K was an input parameter to be set manually. This corresponds to the maximum transient duration in our own work and is automatically discovered.
- Kth-invariants are sequential equivalences that hold after time K, proved with induction which is similar to [14]. We look for transients that are constant after a finite number of time steps and leverage ternary simulation. This means that we likely find fewer transients but are much more scalable. All results presented in this paper were obtained with less than 20 seconds of runtime spent in our proposed simplification steps.

Despite the fact that transients are stronger yet less numerous than Kth-invariants, our approach subsumes [15]. We can time-shift the design by K frames and then pass the design to a general-purpose signal correspondence implementation to find equivalences that hold in all reachable states of the shifted design. This uncovers all Kth-invariants and subsumes the work of [15]. This also explains why simplifying transients and time-shifting the design was beneficial in enhancing the signal correspondence results in Table VI.

Our work is also related to [16] where a sequence of unrolled time frames is used to generate ODC conditions that can be used to simplify the transition relation. [16] asserts that they are simplifying logic that is only used for the initialization of the design, similar to our transients. The prior work was aimed at synthesis and described several exotic scenarios under which the transition relation could be simplified while retaining the correct circuit behavior. Our work differs in that we detect transients and replace them with constants without any need for expensive ODC analysis. We are also aimed at simplifying problems for verification, and in such an environment we do not need to strictly preserve the initialization behavior after it is verified with BMC. However, we do need to time-shift the design to be compatible with other algorithms, a problem that [16] does not address.

VI. Conclusion

This work is concerned with two types of redundant information present in RT-level designs: transient signals and initialization inputs. Algorithms to identify and remove both phenomena were presented, and these algorithms were effective in efficiently simplifying our designs.

The proposed algorithms were implemented in an industrial verification environment as a light-weight design simplification step prior to the invocation of heavy-weight formal verification algorithms. This setup allowed us to examine the relationship of our techniques to existing synthesis and verification algorithms in detail:

- Our techniques increase the number of time frames checked by BMC in a 30-minute timeout by 2.4%.
- Many of the safety properties in the designs we examined were not provable by interpolation. After our simplifications an additional 13.5% of these properties were provable with interpolation.
- Applying our simplifications before signal correspondence allowed for a 0.9% better reduction in Ands and 1.7% better reduction in Registers.
- Our simplifications were effective in simplifying designs after min-register retiming, reducing Ands by 8.1%, Registers by 0.3%, and Inputs by 10.9%.
- Applying our simplifications as a pre-processing step was vital to the completion of an inductive proof of input-output equivalence on large industrial designs.

These results demonstrate that our methods to simplify transient signals and initialization inputs are effective tools in a TBV verification environment.

References

[1] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix, "A Hybrid Verification Approach: Getting Deep into the Design," in *DAC* 2002.

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G/ Janssen, "Scalable Sequential Equivalence Checking across Arbitrary Design Transformations", in *ICCD* 2006.

[3] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner", "Automatic Formal Verification of Fused-Multiply-Add FPUs", in *DATE* 2005.

[4] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV* 2001.

[5] P. Bjesse and J. Kukula, "Automatic Generalized Phase Abstraction for Formal Verification", in *ICCAD* 2005.

[6] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," in *FMSD* 2001.

[7] "Hardware Model Checking Competition 2008: Benchmarks," `http://fmv.jku.at/hwmcc08/benchmarks.html`

[8] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *FMCAD* 2004.

[9] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System," in *TCAD* 1987.

[10] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC 1997*.

[11] A. Mishchenko, S. Chatterjee, and R.K. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC* 2006.

[12] A. Biere, "The AIGER And-Inverter Graph (AIG) Format Version," `http://fmv.jku.at/aiger/FORMAT.aiger`, 2007.

[13] K.L. McMillan, "Interpolation and SAT-Based Model Checking," in *CAV* 2003.

[14] C.A.J. van Eijk, "Sequential equivalence checking based on structural similarities," in *TCAD* 2000.

[15] F. Lu and K.T. Cheng, "Sequential Equivalence Checking Based on K-th Invariants and Circuit SAT Solving," in *HLDVT* 2005.

[16] N. Kitchen and A. Kuehlmann, "Temporal Decomposition for Logic Optimization," in *ICCAD* 2005.

[17] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC* 2006.

Software Model Checking via Large-Block Encoding

Dirk Beyer [*] Simon Fraser University
Alessandro Cimatti [†] FBK-irst, Trento
Alberto Griggio [*‡] University of Trento & Simon Fraser University
M. Erkan Keremoglu [*] Simon Fraser University
Roberto Sebastiani [‡] University of Trento

***Abstract*—Several successful approaches to software verification are based on the construction and analysis of an abstract reachability tree (ART). The ART represents unwindings of the control-flow graph of the program. Traditionally, a transition of the ART represents a single block of the program, and therefore, we call this approach single-block encoding (SBE). SBE may result in a huge number of program paths to be explored, which constitutes a fundamental source of inefficiency. We propose a generalization of the approach, in which transitions of the ART represent larger portions of the program; we call this approach large-block encoding (LBE). LBE may reduce the number of paths to be explored up to exponentially. Within this framework, we also investigate symbolic representations: for representing abstract states, in addition to conjunctions as used in SBE, we investigate the use of arbitrary Boolean formulas; for computing abstract-successor states, in addition to Cartesian predicate abstraction as used in SBE, we investigate the use of Boolean predicate abstraction. The new encoding leverages the efficiency of state-of-the-art SMT solvers, which can symbolically compute abstract large-block successors. Our experiments on benchmark C programs show that the large-block encoding outperforms the single-block encoding.**

I. Introduction

Software model checking is an effective technique for software verification. Several advances in the field have lead to tools that are able to verify programs of considerable size, and show significant advantages over traditional techniques in terms of precision of the analysis (e.g., SLAM [3] and BLAST [6]). However, efficiency and scalability remain major concerns in software model checking and hamper the adaptation of the techniques in industrial practice. Several successful tools for software model checking are based on the construction and analysis of an abstract reachability tree (ART), and predicate abstraction is one of the favorite abstract domains. The ART represents unwindings of the control-flow graph of the program. The search is usually guided by the control flow of the program. Nodes of the ART typically consist of the control-flow location, the call stack, and formulas that represent the data states. During the refinement process, the ART nodes are incrementally refined.

In the traditional ART approach, each program operation (assignment operation, assume operation, function call, function return) is represented by a single edge in the ART. Therefore, we call this approach *single-block encoding* (SBE). A fundamental source of inefficiency of this approach is the fact that the control-flow of the program can induce a huge number of paths (and nodes) in the ART, which are explored independently of each other.

We propose a novel, broader view on ART-based software model checking, where a much more compact abstract space is used, resulting thus in a much smaller number of paths to be enumerated by the ART. Instead of using edges that represent single program operations, we encode entire parts of the program in one edge. In contrast to SBE, we call our new approach *large-block encoding* (LBE). In general, the new encoding may result in an exponential reduction of the number of ART nodes.

The generalization from SBE to LBE has two main consequences. First, LBE requires a more general representation of abstract states than SBE. SBE is typically based on mere *conjunctions* of predicates. Because the LBE approach summarizes large portions of the control flow, conjunctions are not sufficient, and we need to use *arbitrary Boolean combinations* of predicates to represent the abstract states. Second, LBE requires a more accurate abstraction in the abstract-successor computations. Intuitively, an abstract edge represents many different paths of the program, and therefore it is necessary that the abstract-successor computations take the relationships between the predicates into account.

In order to make this generalization practical, we rely on efficient solvers for satisfiability modulo theories (SMT). In particular, enabling factors are the capability of performing Boolean reasoning efficiently (e.g., [21]), the availability of effective algorithms for abstraction computation (e.g., [11], [18]), and interpolation procedures to extract new predicates [9], [12].

Considering Boolean abstraction and large-block encoding in addition to the traditional techniques, we obtain the following interesting observations: (i) whilst the SBE approach requires a large number of successor computations, the LBE approach reduces the number of successor computations dramatically (possibly exponentially); (ii) whilst Cartesian abstraction can be efficiently computed with a linear number of SMT solver queries, Boolean abstraction is expensive to compute because it requires an enumeration of all satisfiable assignments for the predicates. Therefore, two combinations of the above strategies provide an interesting tradeoff: The combination of SBE with Cartesian abstraction was successfully implemented by tools like BLAST and SLAM. We investigate the combination of LBE with Boolean abstraction, by first

[*] Supported in part by the Canadian NSERC grant RGPIN 341819-07 and by the SFU grant PRG 06-3. [†] Supported in part by the European Commission under project FP7-2007-IST-1-217069 COCONUT. [‡] Supported in part by SRC/GRC under Custom Research Project 2009-TJ-1880 WOLFLING and by MIUR under PRIN project 20079E5KM8_002.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

formally defining LBE in terms of a summarization of the control-flow automaton for the program, and then implementing this LBE approach together with a Boolean predicate abstraction. We evaluate the performance and precision by comparing it with the model checker BLAST and with an own implementation of the traditional approach. Our own implementation of the SBE and LBE approach is integrated as a new component into CPACHECKER [8][1]. The experiments show that, despite the simplicity of the idea underlying LBE, our new approach outperforms the previous approach.

Example. We illustrate the advantage of LBE over SBE on the example program in Fig. 1 (a). In SBE, each program location is modeled explicitly, and an abstract-successor computation is performed for each program operation. Figure 1 (b) shows the structure of the resulting ART. In the figure, abstract states are drawn as ellipses, and labeled with the location of the abstract state; the arrows indicate that there exists an edge from the source location to the target location in the control-flow. The ART represents all feasible program paths. For example, the leftmost program path is taking the 'then' branch of every 'if' statement. For every edge in the ART, an abstract-successor computation is performed, which potentially includes several SMT solver queries. The problems given to the SMT solver are usually very small, and the runtime sums up over a large amount of simple queries. Therefore, model checkers that are based on SBE (like BLAST) experience serious performance problems on programs with such an exploding structure (cf. the `test_locks` examples in Table I). In LBE, the control-flow graph is summarized, such that control-flow edges represent entire subgraphs of the original control-flow. In our example, most of the program is summarized into one control-flow edge. Figure 1 (c) shows the structure of the resulting ART, in which all feasible paths of the program are represented by one single edge. The exponential growth of the ART does not occur. □

Related Work. The model checkers SLAM and BLAST are typical examples for the SBE approach [3], [6], both based on counterexample-guided abstraction refinement (CEGAR) [13]. The tool SATABS is also based on CEGAR, but it performs a fully symbolic search in the abstract space [15]. In contrast, our approach still follows the lazy-abstraction paradigm [17], i.e., it abstracts and refines chunks of the program "on-the-fly". The work of McMillan is also based on lazy abstraction, but instead of using predicate abstraction as abstract domain, he directly uses Craig interpolants from infeasible error paths, thus avoiding abstract-successor computations [19]. A different approach to software model checking is bounded model checking (BMC), with the most prominent example CBMC [14]. Programs are unrolled up to a given depth, and a formula is constructed which is satisfiable iff one of the considered program executions reaches a certain error location. The BMC approaches are targeted towards discovering bugs, and can not be used to prove program safety. Finally, the summarizations performed in our large-block encoding bear some similarities with the generation of verification conditions as performed by static program verifiers like SPEC# [4] or CALYSTO [1].

Structure. Section II provides the necessary background. Section III explains our contribution in detail. We experimentally evaluate our novel approach in Sect. IV. In Sect. V, we draw some conclusions and outline directions for future research.

II. BACKGROUND

A. Programs and Control-Flow Automata

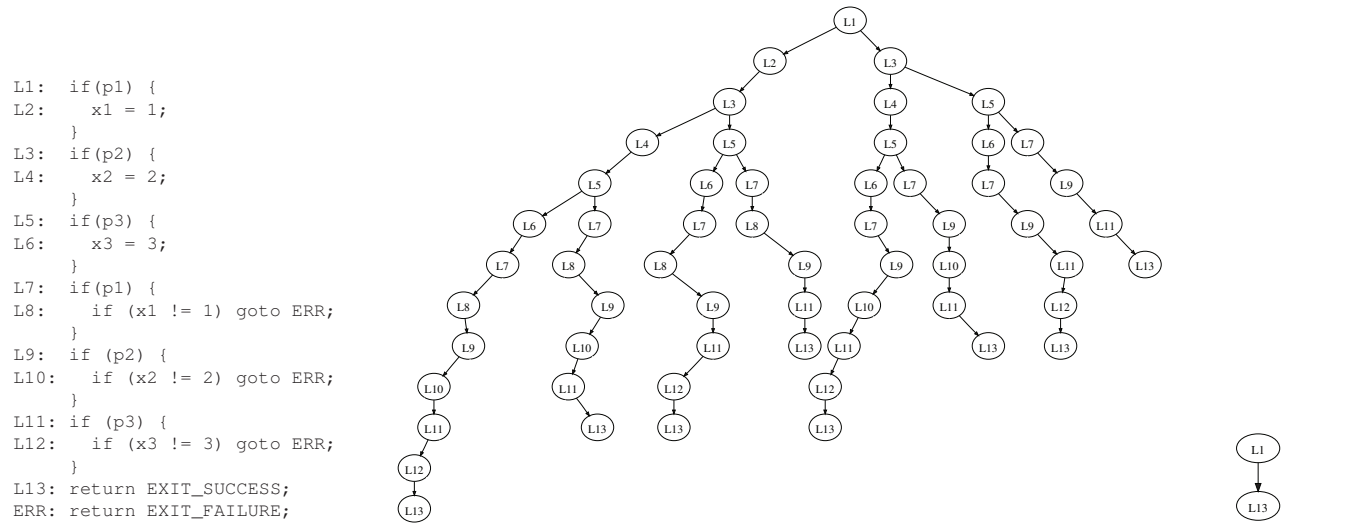
We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[2] We represent a program by a *control-flow automaton* (CFA). A CFA $A = (L, G)$ consists of a set L of program locations, which model the program counter l, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to another. The set of variables that occur in operations from Ops is denoted by X. A *program* $P = (A, l_0, l_E)$ consists of a CFA $A = (L, G)$ (which models the control flow of the program), an initial program location $l_0 \in L$ (which models the program entry) such that G does not contain any edge $(\cdot, \cdot, l_0)$, and a target program location $l_E \in L$ (which models the error location).

A *concrete data state* of a program is a variable assignment $c : X \to \mathbb{Z}$ that assigns to each variable an integer value. The set of all concrete data states of a program is denoted by $\mathcal{C}$. A set $r \subseteq \mathcal{C}$ of concrete data states is called *region*. We represent regions using first-order formulas (with free variables from X): a formula φ represents the set S of all data states c that imply φ (i.e., $S = \{c \mid c \models \varphi\}$). A *concrete state* of a program is a pair (l, c), where $l \in L$ is a program location and c is a concrete data state. A pair (l, φ) represents the following set of all concrete states: $\{(l, c) \mid c \models \varphi\}$. The *concrete semantics* of an operation $op \in Ops$ is defined by the strongest postcondition operator SP_{op}: for a formula φ, $\mathsf{SP}_{op}(\varphi)$ represents the set of data states that are reachable from any of the states in the region represented by φ after the execution of op. Given a formula φ that represents a set of concrete data states, for an assignment operation $s := e$, we have $\mathsf{SP}_{s:=e}(\varphi) = \exists \widehat{s} : \varphi_{[s \mapsto \widehat{s}]} \wedge (s = e_{[s \mapsto \widehat{s}]})$; and for an assume operation $assume(p)$, we have $\mathsf{SP}_{assume(p)}(\varphi) = \varphi \wedge p$.

A *path* σ is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of operations and locations. The path σ is called *program path* if for every i with $1 \leq i \leq n$ there exists a CFA edge $g = (l_{i-1}, op_i, l_i)$, i.e., σ represents a syntactical walk through the CFA. The *concrete semantics for a program path* $\sigma = \langle (op_1, l_1), ..., (op_n, l_n) \rangle$ is defined as the successive application of the strongest postoperator for each operation: $\mathsf{SP}_\sigma(\varphi) = \mathsf{SP}_{op_n}(...\mathsf{SP}_{op_1}(\varphi)...)$. The set of concrete states that result from running σ is represented by the pair $(l_n, \mathsf{SP}_\sigma(true))$. A program path σ is *feasible* if

[1] Available at `http://www.sosy-lab.org/~dbeyer/CPAchecker`

[2] Our implementation is based on CPACHECKER, which operates on C programs that are given in the CIL intermediate language [20]; function calls are supported.



(a) Example C program (b) ART for SBE (c) ART for LBE

Fig. 1. Example program and corresponding ARTs for SBE and LBE; this example was considered as verification challenge for ART-based approaches

$\mathsf{SP}_\sigma(true)$ is satisfiable. A concrete state (l_n, c_n) is called *reachable* if there exists a feasible program path σ whose final location is l_n and such that $c_n \models \mathsf{SP}_\sigma(true)$. A location l is reachable if there exists a concrete state c such that (l, c) is reachable. A program is *safe* if l_E is not reachable.

B. Predicate Abstraction

Let $\mathcal{P}$ be a set of predicates over program variables in a quantifier-free theory $\mathcal{T}$. A *formula* φ is a Boolean combination of predicates from $\mathcal{P}$. A *precision for a formula* is a finite subset $\pi \subset \mathcal{P}$ of predicates.

Cartesian Predicate Abstraction. Let π be a precision. The *Cartesian predicate abstraction* $\varphi^\pi_\mathbb{C}$ *of a formula* φ is the strongest conjunction of predicates from π that is entailed by φ: $\varphi^\pi_\mathbb{C} := \bigwedge \{p \in \pi \mid \varphi \Rightarrow p\}$. Such a predicate abstraction of a formula φ, which represents a region of concrete program states, is used as an *abstract state* (i.e., an abstract representation of the region) in program verification. For a formula φ and a precision π, the Cartesian predicate abstraction $\varphi^\pi_\mathbb{C}$ of φ can be computed by $|\pi|$ SMT-solver queries. The abstract strongest postoperator SP^π for a predicate abstraction with precision π transforms the abstract state $\varphi^\pi_\mathbb{C}$ into its successor ${\varphi'}^\pi_\mathbb{C}$ for a program operation op, written as ${\varphi'}^\pi_\mathbb{C} = \mathsf{SP}^\pi_{op}(\varphi^\pi_\mathbb{C})$, if ${\varphi'}^\pi_\mathbb{C}$ is the Cartesian predicate abstraction of $\mathsf{SP}_{op}(\varphi^\pi_\mathbb{C})$, i.e., ${\varphi'}^\pi_\mathbb{C} = (\mathsf{SP}_{op}(\varphi^\pi_\mathbb{C}))^\pi_\mathbb{C}$. For more details, we refer the reader to the work of Ball et al. [2].

Boolean Predicate Abstraction. Let π be a precision. The *Boolean predicate abstraction* $\varphi^\pi_\mathbb{B}$ *of a formula* φ is the strongest Boolean combination of predicates from π that is entailed by φ. For a formula φ and a precision π, the Boolean predicate abstraction $\varphi^\pi_\mathbb{B}$ of φ can be computed by querying an SMT solver in the following way: For each predicate $p_i \in \pi$, we introduce a propositional variable v_i. Now we ask an SMT solver to enumerate all satisfying assignments of $v_1, ..., v_{|\pi|}$ in the formula $\varphi \wedge \bigwedge_{p_i \in \pi}(p_i \Leftrightarrow v_i)$. For each satisfying assignment, we construct a conjunction of all predicates from π whose corresponding propositional variable occurs positive in the assignment. The disjunction of all such conjunctions is the Boolean predicate abstraction for φ. The abstract strongest postoperator SP^π for a predicate abstraction with precision π transforms the abstract state $\varphi^\pi_\mathbb{B}$ into its successor ${\varphi'}^\pi_\mathbb{B}$ for a program operation op, written as ${\varphi'}^\pi_\mathbb{B} = \mathsf{SP}^\pi_{op}(\varphi^\pi_\mathbb{B})$, if ${\varphi'}^\pi_\mathbb{B}$ is the Boolean predicate abstraction of $\mathsf{SP}_{op}(\varphi^\pi_\mathbb{B})$, i.e., ${\varphi'}^\pi_\mathbb{B} = (\mathsf{SP}_{op}(\varphi^\pi_\mathbb{B}))^\pi_\mathbb{B}$. For more details, we refer the reader to the work of Lahiri et al. [18].

C. ART-based Software Model Checking with SBE

The *precision for a program* is a function $\Pi : L \to 2^\mathcal{P}$, which assigns to each program location a precision for a formula. An ART-based algorithm for software model checking takes an initial precision Π (which is typically very coarse) for the predicate abstraction, and constructs an ART for the input program and Π. An ART is a tree whose nodes are labeled with program locations and abstract states [6] (i.e., $n = (l, \varphi)$). For a given ART node, all children nodes are labeled with successor locations and abstract successor states, according to the strongest postoperator and the predicate abstraction. A node $n = (l, \varphi)$ is called *covered* if there exists another ART node $n' = (l, \varphi')$ that entails n (i.e., s.t. $\varphi' \models \varphi$). An ART is called *complete* if every node is either covered or all possible abstract successor states are present in the ART as children of the node. If a complete ART is constructed and the ART does not contain any error node, then the program is considered correct [6]. If the algorithm adds an error node to the ART, then the corresponding path σ is checked to determine if σ is feasible (i.e., if the corresponding concrete program path is executable) or infeasible (i.e., if there is no corresponding program execution). In the former case the path represents a witness for a program bug. In the latter case the path is analyzed, and a refinement Π' of Π is generated, such that the same path cannot occur again during the ART exploration. The concept of using an infeasible error path for abstraction refine-

ment is called counterexample-guided abstraction refinement (CEGAR) [13]. The concept of iteratively constructing an ART and refining only the precisions along the considered path is called lazy abstraction [17]. Craig interpolation is a successful approach to predicate extraction during refinement [16]. After refining the precision, the algorithm continues with the next iteration, using Π' instead of Π to construct the ART, until either a complete error-free ART is obtained, or an error is found (note that the procedure might not terminate). For more details and a more in-depth illustration of the overall ART algorithm, we refer the reader to the BLAST article [6].

In order to make the algorithm scale on practical examples, implementations such as BLAST or SLAM use the simple but coarse Cartesian abstraction, instead of the expensive but precise Boolean abstraction. Despite its potential imprecision, Cartesian abstraction has been proved successful for the verification of many real-world programs. In the SBE approach, given the large number of successor computations, the computation of the Boolean predicate abstraction is in fact too expensive, as it may require an SMT solver to enumerate an exponential number of assignments on the predicates in the precision, for each single successor computation. The reason for the success of Cartesian abstraction if used together with SBE, is that for a given program path, state over-approximations that are expressible as conjunctions of atomic predicates —for which Boolean and Cartesian abstractions are equivalent— are often good enough to prove that the error location is not reachable in the abstract space.

III. LARGE-BLOCK ENCODING

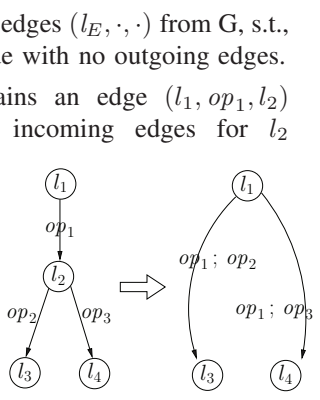
A. Summarization of Control-Flow Automata

The large-block encoding is achieved by a summarization of the program CFA, in which each loop-free subgraph of the CFA is replaced by a single control-flow edge with a large formula that represents the removed subgraph. This process, which we call *CFA-summarization*, consists of the fixpoint application of the three rewriting rules that we describe below: first we apply Rule 0 once, and then we repeatedly apply Rules 1 and 2, until no rule is applicable anymore.

Let $P = (A, l_0, l_E)$ be a program with CFA $A = (L, G)$.

Rule 0 (Error Sink). We remove all edges $(l_E, \cdot, \cdot)$ from G, s.t., the target location l_E is a sink node with no outgoing edges.

Rule 1 (Sequence). If G contains an edge (l_1, op_1, l_2) with $l_1 \neq l_2$ and no other incoming edges for l_2 (i.e. edges $(\cdot, \cdot, l_2)$), and $G^{\rightarrow}_{l_2}$ is the subset of G of outgoing edges for l_2, then we change the CFA A in the following way: (1) we remove location l_2 from L, and (2) we remove the edge (l_1, op_1, l_2) and all edges in $G^{\rightarrow}_{l_2}$ from G, and for each edge $(l_2, op_i, l_i) \in G^{\rightarrow}_{l_2}$, we add the edge $(l_1, op_1 \,;\, op_i, l_i)$ to G, where $\mathsf{SP}_{op_1 \,;\, op_i}(\varphi) = \mathsf{SP}_{op_i}(\mathsf{SP}_{op_1}(\varphi))$. (Note that $G^{\rightarrow}_{l_2}$ might contain an edge $(l_2, \cdot, l_1)$.)



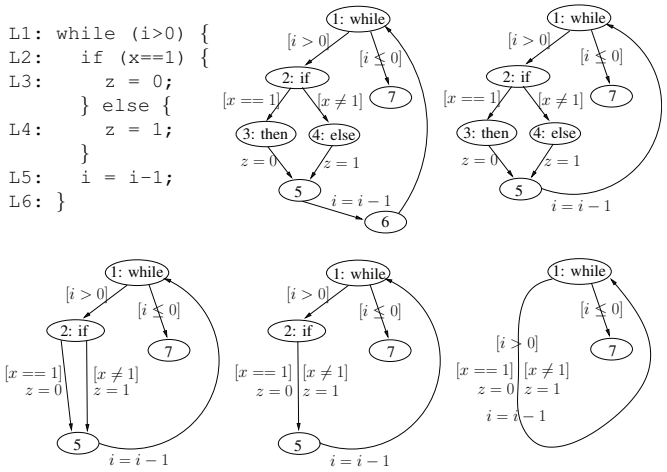
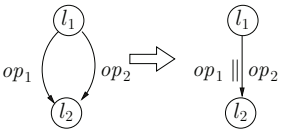


Fig. 2. CFA Transformation: a) Program, b) CFA, c)–e) Intermediate CFAs, f) CFA-Summary. In the CFAs, $assume(p)$ is represented as $[p]$, $op_1 \,;\, op_2$ is represented by drawing op_2 below op_1, and $op_1 \parallel op_2$ by drawing op_2 beside op_1

Rule 2 (Choice). If $L_2 = \{l_1, l_2\}$ and $A_{|L_2} = (L_2, G_2)$ is the subgraph of A with nodes from L_2 and the set G_2 of edges contains the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2), then we change the CFA A in the following way: (1) we remove the two edges (l_1, op_1, l_2) and (l_1, op_2, l_2) from G and add the edge $(l_1, op_1 \parallel op_2, l_2)$ to G, where $\mathsf{SP}_{op_1 \parallel op_2}(\varphi) = \mathsf{SP}_{op_1}(\varphi) \vee \mathsf{SP}_{op_2}(\varphi)$. (Note that there might be a backwards edge $(l_2, \cdot, l_1)$.)



Let $P = (A, l_0, l_E)$ be a program and let A' be a CFA. The CFA A' is a *CFA-summary* of A if A' is obtained from A via application of Rule 1 and then stepwise application of Rules 1 and 2, and no rule can be further applied.

Example. Figure 2 shows a program (a) and its corresponding CFA (b). The control-flow automaton (CFA) is iteratively transformed to a CFA-summary (f) as follows: Rule 1 eliminates location 6 to (c), Rule 1 eliminates location 3 and then location 4 to (d), Rule 2 replaces the two edges 2–5 to (e), Rule 1 eliminates location 2 and then location 5 to (f). □

In the context of this article, we use the CFA-summary for program analysis, i.e., we want to verify if the error location of the program is reachable. The following theorem states that our summarization of a CFA is correct in this sense. (The proof can be found in our extended technical report [5].)

Theorem 3.1 (Correctness of Summarization): Let $P = (A, l_0, l_E)$ be a program and let $A' = (L', G')$ be a CFA-summary of A. Then: (i) $\{l_0, l_E\} \subseteq L'$, and (ii) l_E is reachable in (A', l_0, l_E) if and only if l_E is reachable in P.

The summarization can be performed in polynomial time. The time taken by Rule 0 is proportional to the number of outgoing edges for l_E. Since each application of Rule 1 or Rule 2 removes at least one edge, there can be at most $|G| - 1$ such applications. A naive way to determine the set of locations and edges to which to apply each rule requires

$O(|V| \cdot k)$ time, where k is the maximum out-degree of locations. Finally, each application of Rule 2 requires $O(1)$ time, and each application of Rule 1 $O(k)$ time. Therefore, a naive summarization algorithm requires $O(|G| \cdot |V| \cdot k)$ time, which reduces to $O(|G| \cdot |V|)$ if k is bounded (i.e., if we rewrite a priori all `switch`es into nested `if`s).[3]

B. LBE versus SBE for Software Model Checking

The use of LBE instead of the standard SBE requires no modification to the general model-checking algorithm, which is still based on ART construction with CEGAR-based refinement. The main difference is that in LBE, there is no one-to-one correspondence between ART paths and syntactical program paths. A single CFA edge corresponds to a *set of paths* between its source and target location, and a single ART path corresponds to a *set of program paths*. An ART node represents an overapproximation of the data region that is reachable by following *any* of the program paths represented by the ART path that leads to it. This difference leads to two observations.

First, LBE can lead to exponentially-smaller ARTs than SBE, and thus it can drastically reduce the number of successor computations (cf. example in Sect. I) and the number of abstraction-refinement steps for infeasible error paths. Each of these operations, however, is typically more expensive than with SBE, because more complex formulas are involved.

Second, LBE requires a more general representation of abstract states. When using SBE, abstract states are typically represented as sets/conjunctions of predicates. This is sufficient for practical examples because each abstract state represents a data region reachable by a single program path, which can be encoded essentially as a conjunction of atomic formulas. With LBE, such representation would be too coarse, since each abstract state represents a data region that is reachable on several different program paths. Therefore, we need to use a representation for arbitrary (and larger) Boolean combinations of predicates. This generalization of the representation of abstract states requires a generalization of the representation of the transfers, i.e., replacing the Cartesian abstraction with a more precise form of abstraction. In this paper, we evaluate the use of the Boolean abstraction, which allows for a precise representation of arbitrary Boolean combinations of predicates.

With respect to the traditional SBE approach, LBE allows us to trade part of the cost of the *explicit* enumeration of program paths with that of the *symbolic* computation of abstract successor states: rather than having to build large ARTs via SBE by performing a substantial amount of relatively cheap operations (Cartesian abstract postoperator applications along single-block edges and counterexample analysis of individual program paths), we build smaller ARTs via LBE by performing more expensive symbolic operations (Boolean abstract postoperator applications along large portions of the control flow and counterexample analysis of multiple program paths), involving formulas with a complex Boolean structure. With LBE, the *cost* of each symbolic operation, rather than their *number*, becomes a critical performance factor.

To this extent, LBE makes it possible to fully exploit the power and functionality of modern SMT solvers: First, the capability of modern SMT solvers to perform large amounts of Boolean reasoning allows for handling large Boolean combinations of atomic expressions, instead of simple conjunctions. Second, the capability of some SMT solvers (e.g., [10]) to perform All-SMT and interpolation allows for efficient computation of Boolean abstractions and interpolants, respectively. SMT-based Boolean abstraction and interpolation were shown to outperform previous approaches [11], [12], [18], especially when dealing with complex formulas. With SBE, instead, the use of modern SMT technology does not lead to significant improvements of the overall ART-based algorithm, because each SMT query involves only simple conjunctions.[4]

IV. Performance Evaluation

Implementation. In order to evaluate the proposed verification method, we integrate our algorithm as a new component into the configurable software verification toolkit CPAchecker [8]. This implementation is written in Java. All example programs are preprocessed and transformed into the simple intermediate language CIL [20]. For parsing C programs, CPAchecker uses a library from the Eclipse C/C++ Development Kit. For efficient querying of formulas in the quantifier-free theory of rational linear arithmetic and equality with uninterpreted function symbols, we use the SMT solver MathSAT [10], which is integrated as a library (written in C++). We use BDDs for representing abstract-state formulas.

Our benchmark programs, the source code, and an executable of our LBE implementation are available on the supplementary web site on Large-Block Encoding (`http://www.sosy-lab.org/~dbeyer/cpa-lbe`). We ran all experiments on a 1.8 GHz Intel Core2 machine with 2 GB of RAM and 2 MB of cache, running GNU/Linux. We used a timeout of 1 800 s and a memory limit of 1.8 GB.

Example Programs. We use three categories of benchmark programs. First, we experiment with programs that are specifically designed to cause an exponential blowup of the ART when using SBE (`test_locks*`, in the style of the example in Sect. I). Second, we use the device-driver programs that were previously used as benchmarks in the Blast project.[5] Third, we solve various verification problems for the SSH client

[3] In our implementation, we use a more efficient algorithm, which we do not describe here for lack of space.

[4] For example, Blast uses Simplify, version 1.5.4, as of October 2001, for computing abstract successor states. Experiments have shown that replacing this old Simplify version by a highly-tuned modern SMT solver does not significantly improve the performance, because Blast does not use much power of the SMT solver. Moreover, it was shown that although the MathSAT SMT solver outperformed other tools in the computation of Craig interpolants for general formulas, the difference in performance is negligible on formulas generated by a standard SBE ART-based algorithm [12].

[5] The Blast distribution contains 8 Windows driver benchmarks. However, we could not run three of them (`parclass.i`, `mouclass.i`, and `serial.i`), because CIL fails to parse them, making both CPAchecker and Blast fail.

TABLE I
PERFORMANCE RESULTS

Program	BLAST (best result)	CPACHECKER SBE	CPACHECKER LBE
test_locks_5.c	4.50	4.01	**0.29**
test_locks_6.c	7.81	7.22	**0.32**
test_locks_7.c	13.91	12.63	**0.34**
test_locks_8.c	25.00	23.93	**0.57**
test_locks_9.c	46.84	52.04	**0.38**
test_locks_10.c	94.57	131.39	**0.40**
test_locks_11.c	204.55	MO	**0.70**
test_locks_12.c	529.16	MO	**0.46**
test_locks_13.c	1229.27	MO	**0.49**
test_locks_14.c	>1800.00	MO	**0.50**
test_locks_15.c	>1800.00	MO	**0.56**
cdaudio.i.cil.c	175.76	MO	**53.55**
diskperf.i.cil.c	>1800.00	MO	**232.00**
floppy.i.cil.c	218.26	MO	**56.36**
kbfiltr.i.cil.c	23.55	41.12	**7.82**
parport.i.cil.c	738.82	MO	**378.04**
s3_clnt.blast.01.i.cil.c	33.01	755.81	**19.51**
s3_clnt.blast.02.i.cil.c	62.65	1075.45	**16.00**
s3_clnt.blast.03.i.cil.c	60.62	746.31	**49.50**
s3_clnt.blast.04.i.cil.c	63.96	730.80	**25.45**
s3_srvr.blast.01.i.cil.c	811.27	>1800.00	**125.33**
s3_srvr.blast.02.i.cil.c	360.47	>1800.00	**122.83**
s3_srvr.blast.03.i.cil.c	276.19	>1800.00	**98.47**
s3_srvr.blast.04.i.cil.c	175.64	>1800.00	**71.77**
s3_srvr.blast.06.i.cil.c	304.63	>1800.00	**59.70**
s3_srvr.blast.07.i.cil.c	478.05	>1800.00	**85.82**
s3_srvr.blast.08.i.cil.c	115.76	>1800.00	**61.29**
s3_srvr.blast.09.i.cil.c	445.21	>1800.00	**126.47**
s3_srvr.blast.10.i.cil.c	115.10	>1800.00	**63.36**
s3_srvr.blast.11.i.cil.c	367.98	>1800.00	**162.76**
s3_srvr.blast.12.i.cil.c	304.05	>1800.00	**170.33**
s3_srvr.blast.13.i.cil.c	580.33	>1800.00	**74.49**
s3_srvr.blast.14.i.cil.c	303.21	>1800.00	**50.38**
s3_srvr.blast.15.i.cil.c	115.88	>1800.00	**21.01**
s3_srvr.blast.16.i.cil.c	305.11	>1800.00	**127.82**
TOTAL (solved/time)	**32 / 8591.12**	**11 / 3580.71**	**35 / 2265.07**
TOTAL w/o `test_locks*`	**23 / 6435.51**	**5 / 3349.48**	**24 / 2260.07**

and server software (`s3_clnt*` and `s3_srvr*`), which share the same program logic, but check different safety properties. The safety property is encoded as conditional call of a failure location and therefore reduces to the reachability of a certain error location. All benchmark programs from the BLAST web page are preprocessed with CIL. For the second and third groups of programs, we also performed experiments with artificial defects introduced.

Experimental Configurations. For a careful and fair performance comparison, we performed experiments using three different configurations. First, we use BLAST, version 2.5, which is a highly optimized state-of-the-art software model checker. BLAST is implemented in the programming language OCAML. We ran BLAST using all four combinations of breadth-first search (`-bfs`) versus depth-first search (`-dfs`), both with and without heuristics for improving the predicate discovery. BLAST provides five different levels of heuristics for predicate discovery, and we use only the lowest (`-predH 0`) and the highest option (`-predH 7`). Interestingly, every combination is best for some particular example programs, with considerable differences in runtime and memory consumption.

TABLE II
PERFORMANCE RESULTS, PROGRAMS WITH ARTIFICIAL BUGS

Program	BLAST (best result)	CPACHECKER SBE	CPACHECKER LBE
cdaudio.BUG.i.cil.c	18.79	74.39	**9.85**
diskperf.BUG.i.cil.c	889.79	26.53	**6.78**
floppy.BUG.i.cil.c	119.60	36.49	**4.30**
kbfiltr.BUG.i.cil.c	46.80	75.45	**11.52**
parport.BUG.i.cil.c	**1.67**	14.62	2.64
s3_clnt.blast.01.BUG.i.cil.c	8.84	1514.90	**3.33**
s3_clnt.blast.02.BUG.i.cil.c	9.02	843.42	**3.27**
s3_clnt.blast.03.BUG.i.cil.c	6.64	780.72	**2.61**
s3_clnt.blast.04.BUG.i.cil.c	9.78	724.04	**3.18**
s3_srvr.blast.01.BUG.i.cil.c	7.59	MO	**2.09**
s3_srvr.blast.02.BUG.i.cil.c	7.16	>1800.00	**2.10**
s3_srvr.blast.03.BUG.i.cil.c	7.42	>1800.00	**2.08**
s3_srvr.blast.04.BUG.i.cil.c	7.33	>1800.00	**1.93**
s3_srvr.blast.06.BUG.i.cil.c	39.81	MO	**5.08**
s3_srvr.blast.07.BUG.i.cil.c	310.84	>1800.00	**28.35**
s3_srvr.blast.08.BUG.i.cil.c	40.51	>1800.00	**36.47**
s3_srvr.blast.09.BUG.i.cil.c	265.48	>1800.00	**4.94**
s3_srvr.blast.10.BUG.i.cil.c	40.24	>1800.00	**12.01**
s3_srvr.blast.11.BUG.i.cil.c	49.05	>1800.00	**4.80**
s3_srvr.blast.12.BUG.i.cil.c	38.66	>1800.00	**6.11**
s3_srvr.blast.13.BUG.i.cil.c	251.56	>1800.00	**15.20**
s3_srvr.blast.14.BUG.i.cil.c	39.94	1656.54	**4.63**
s3_srvr.blast.15.BUG.i.cil.c	40.19	>1800.00	**10.19**
s3_srvr.blast.16.BUG.i.cil.c	39.54	>1800.00	**5.21**
TOTAL (solved/time)	**24 / 2296.25**	**10 / 5747.10**	**24 / 188.67**

The configuration using `-dfs -predH 7` is the winner (in terms of solved problems and total runtime) for the programs without defects, but is not able to verify four example programs (timeout) [5]. For the unsafe programs, `-bfs -predH 7` performs best. All four configurations use the command-line options `-craig 2 -nosimplemem -alias ""`, which specify that BLAST runs with lazy, Craig-interpolation-based refinement, no CIL preprocessing for memory access, and without pointer analysis. In all experiments with BLAST, we use the same interpolation procedure (MATHSAT) as in our CPACHECKER-based implementation.[6] In the performance tables, we show the best result among the four configurations for every single instance (column `best result`). (The results of all four configurations are provided in our extended technical report [5].)

Second, in order to separate the optimization efforts in BLAST from the conceptual essence of the traditional lazy-abstraction algorithm, we developed a re-implementation of the traditional algorithms (column 'SBE'), as described in the BLAST tool article [6]. This re-implementation is integrated as component into CPACHECKER, so that the difference between SBE and LBE is only in the algorithms, not in the environment (same parser, same BDD package, same query optimization, etc.). Our SBE implementation uses a DFS algorithm.

Third, we ran the experiments using our new LBE algorithm, which is also implemented within CPACHECKER (column LBE). Our LBE implementation uses a DFS algorithm. Note that the purpose of our experiments is to give evidence of the performance difference between SBE and LBE, because these two settings are closest to each other, since SBE and LBE

[6]We tried also to use MATHSAT instead of SIMPLIFY for computing abstract successor states, but this did not improve the performance of BLAST.

TABLE III
DETAILED COMPARISON BETWEEN SBE AND LBE; ENTRIES MARKED WITH (*) DENOTE PARTIAL STATISTICS FOR ANALYSES THAT TERMINATED UNSUCCESSFULLY (IF AVAILABLE)

	SBE					LBE				
	ART	# ref	# predicates			ART	# ref	# predicates		
Program	size	steps	Tot	Avg	Max	size	steps	Tot	Avg	Max
test_locks_5.c	1344	50	10	3	10	4	0	0	0	0
test_locks_6.c	2301	72	12	4	12	4	0	0	0	0
test_locks_7.c	3845	98	14	5	14	4	0	0	0	0
test_locks_8.c	6426	128	16	6	16	4	0	0	0	0
test_locks_9.c	10926	162	18	7	18	4	0	0	0	0
test_locks_10.c	19091	200	20	8	20	4	0	0	0	0
test_locks_11.c	24779(*)	242(*)	22(*)	9(*)	22(*)	4	0	0	0	0
test_locks_12.c	28119(*)	288(*)	24(*)	10(*)	24(*)	4	0	0	0	0
test_locks_13.c	31739(*)	338(*)	26(*)	10(*)	26(*)	4	0	0	0	0
test_locks_14.c	35178(*)	392(*)	28(*)	11(*)	28(*)	4	0	0	0	0
test_locks_15.c	38777(*)	450(*)	30(*)	12(*)	30(*)	4	0	0	0	0
cdaudio.i.cil.c	53323(*)	445(*)	147(*)	9(*)	78(*)	6909	140	79	5	16
diskperf.i.cil.c	–	–	–	–	–	4890	145	56	6	21
floppy.i.cil.c	31079(*)	301(*)	79(*)	7(*)	35(*)	9668	176	58	4	13
kbfiltr.i.cil.c	19640	153	53	5	27	1577	47	18	2	6
parport.i.cil.c	26188(*)	360(*)	143(*)	4(*)	41(*)	38488	474	168	4	17
s3_clnt.blast.01.i.cil.c	122678	557	59	20	59	36	5	47	11	47
s3_clnt.blast.02.i.cil.c	354132	532	55	19	55	36	5	51	12	51
s3_clnt.blast.03.i.cil.c	196599	534	55	19	55	39	5	75	18	75
s3_clnt.blast.04.i.cil.c	172444	538	55	19	55	36	5	47	11	47
s3_srvr.blast.01.i.cil.c	232195(*)	774(*)	70(*)	20(*)	70(*)	101	6	88	22	88
s3_srvr.blast.02.i.cil.c	254667(*)	745(*)	79(*)	19(*)	78(*)	109	7	75	18	75
s3_srvr.blast.03.i.cil.c	–	–	–	–	–	91	6	85	21	85
s3_srvr.blast.04.i.cil.c	–	–	–	–	–	103	7	82	20	82
s3_srvr.blast.06.i.cil.c	295698(*)	576(*)	63(*)	14(*)	63(*)	94	6	84	21	84
s3_srvr.blast.07.i.cil.c	–	–	–	–	–	92	5	85	21	85
s3_srvr.blast.08.i.cil.c	279991(*)	549(*)	57(*)	15(*)	57(*)	89	5	88	22	88
s3_srvr.blast.09.i.cil.c	189541(*)	720(*)	72(*)	16(*)	71(*)	193	4	72	18	72
s3_srvr.blast.10.i.cil.c	307671(*)	597(*)	55(*)	16(*)	55(*)	91	5	79	19	79
s3_srvr.blast.11.i.cil.c	–	–	–	–	–	48	6	69	17	69
s3_srvr.blast.12.i.cil.c	258546(*)	563(*)	57(*)	15(*)	57(*)	99	6	94	23	94
s3_srvr.blast.13.i.cil.c	167333(*)	682(*)	70(*)	18(*)	69(*)	90	5	81	20	81
s3_srvr.blast.14.i.cil.c	318982(*)	643(*)	65(*)	13(*)	64(*)	92	6	83	20	83
s3_srvr.blast.15.i.cil.c	279319(*)	579(*)	58(*)	15(*)	58(*)	71	4	71	17	71
s3_srvr.blast.16.i.cil.c	346185(*)	596(*)	59(*)	12(*)	58(*)	98	6	86	21	86

differ only in the CFA summarization and Boolean abstraction. The first column is provided in Tables I and II to give evidence that the new approach beats the highly-optimized traditional implementation BLAST.

We actually configured and ran experiments with all four combinations: SBE versus LBE, and Cartesian versus Boolean abstraction. The experimentation clearly showed that SBE does not benefit from Boolean abstraction in terms of precision, with substantial degrade in performance: the only programs for which it terminated successfully were the first five instances of the `test_locks` group. Similarly, the combination of LBE with Cartesian abstraction fails to solve any of the experiments, due to loss of precision. Thus, we report only on the two successful configurations, i.e., SBE in combination with Cartesian abstraction, and LBE with Boolean abstraction.

Discussion of Evaluation Results. Tables I and II present performance results of our experiments, for the safe and unsafe programs respectively. All runtimes are given in seconds of processor time, '>1800.00' indicates a timeout, 'MO' indicates an out-of-memory. Table III shows statistics about the algorithms for SBE and LBE only.

The first group of experiments in Table I shows that the time complexity of SBE (and BLAST) can grow exponentially in the number of nested conditional statements, as expected. Table III explains why the SBE approach exhausts the memory: the number of abstract nodes in the reachability tree grows exponentially in the number of nested conditional statements. Therefore, SBE does not scale. The LBE approach reduces the loop-free part of the branching control-flow structure to a few edges (cf. example in the introduction), and the size of the ART is constant for this example program, because only the structure inside the body of the loop changes. There are no refinement steps necessary in the LBE approach, because the edges to the error location are infeasible. Therefore, no predicates are used. The runtime of the LBE approach slightly increases with the size of the program, because the size of the formulas that are sent to the SMT solver is slightly increasing. Although in principle the complexity of the SMT problem grows exponentially in the size of the formulas, the heuristics used by SMT solvers avoid the exponential enumeration that we observe in the case of SBE.

For the two other classes of experiments, we see that LBE is able to successfully complete all benchmarks, and shows significant performance gains over SBE. SBE is able to solve only about one third of all benchmarks, and for the ones that complete, it is clearly outperformed by LBE. In Table III, we see that SBE has in general a much larger ART. In Table I we observe that LBE performs significantly better than any BLAST configuration. LBE performed best also in finding the error paths (cf. Table II), outperforming both SBE and BLAST.

In summary, the experiments show that the LBE approach outperforms the SBE approach, both for correct and defective programs. This provides evidence of the benefits of a "more symbolic" analysis as performed in the LBE approach. One might argue that our CPACHECKER-based SBE implementation might be sub-optimal although it uses the same implementation and execution environment as LBE; in fact, both implementations currently suffer from some inefficiencies and have room for several optimizations. Therefore, we compare also with BLAST. By looking at Tables I and II, we see that LBE outperforms also BLAST, despite the fact that the latter is the result of several years of fine-tuning. BLAST in turn is much more efficient than SBE. However, the performance gap between BLAST and SBE highly depends on the command-line options used for BLAST.

We conclude the section by discussing the scope of the experimental evaluation. The LBE techniques proposed in this paper bear substantial similarities to the SSA-based encodings used in tools like SATABS [15], CALYSTO [1] or SPEC# [4]. For lack of space, we chose to not include a comparison with such tools; rather, we focussed on the more relevant issue of the impact of LBE on ART-based model checking.

V. CONCLUSION AND FUTURE WORK

We have proposed LBE as an alternative to the SBE model-checking approach, based on the idea that transitions in the abstract space should represent larger fragments of the program. Our novel approach results in significantly smaller ARTs, where abstract successor computations are more involved, and thus trading cost of many explicit enumerations of program paths with the cost of symbolic successor computations. A thorough experimental evaluation shows the advantages of LBE against both our implementation of SBE and the state-of-the-art BLAST system.

The existing experimental results can now be summarized as follows: (i) the combination of Cartesian predicate abstraction with SBE is successful on many practical programs [3], [6], but is not efficient on programs with nested conditional branching, (ii) the combination of Boolean predicate abstraction with SBE is intractably expensive [2], (iii) the combination of Cartesian predicate abstraction with joining paths is too imprecise [7], and (iv) the combination of Boolean predicate abstraction with LBE is the most promising combination for ART-based predicate abstraction (Tables I and II).

In our future work, we plan to implement McMillan's interpolation-based lazy-abstraction approach [19], and experiment with SBE versus LBE versions of his algorithm. Furthermore, we plan to investigate the use of adjustable precision-based techniques for the construction of the large blocks on-the-fly (instead of the current preprocessing step). This would enable a dynamic adjustment of the size of the large blocks, and thus we could fine-tune the amount of work that is delegated to the SMT solver. Also, we plan to explore other techniques for computing abstract successors which are more precise than Cartesian abstraction but less expensive than Boolean abstraction.

Acknowledgments. We thank Roman Manevich for interesting discussions about BLAST's performance bottlenecks.

REFERENCES

[1] D. Babic and A. J. Hu, "CALYSTO: Scalable and precise extended static checking," in *Proc. ICSE*. ACM, 2008, pp. 211–220.

[2] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and cartesian abstractions for model checking C programs," in *Proc. TACAS*, ser. LNCS 2031. Springer, 2001, pp. 268–283.

[3] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proc. POPL*. ACM, 2002, pp. 1–3.

[4] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," in *Proc. PASTE*. ACM, 2005, pp. 82–87.

[5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding, Tech. Rep. SFU-CS-2009-09/DISI-09-026/FBK-irst-2009.04.005, April 2009. Available: http://arxiv.org/abs/0904.4709

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *Int. J. Softw. Tools Technol. Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[7] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," *Proc. CAV*, LNCS 4590. Springer, 2007, pp. 504–518.

[8] D. Beyer and M. E. Keremoglu, "CPACHECKER: A tool for configurable software verification," Simon Fraser University, Tech. Rep. SFU-CS-2009-02, January 2009. Available: http://arxiv.org/abs/0902.0019

[9] D. Beyer, D. Zufferey, and R. Majumdar, "CSISAT: Interpolation for LA+EUF," in *Proc. CAV*, LNCS 5123. Springer, 2008, pp. 304–308.

[10] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, "The MATHSAT 4 SMT solver," in *Proc. CAV*, ser. LNCS 5123. Springer, 2008, pp. 299–303.

[11] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar, "Computing predicate abstractions by integrating BDDs and SMT solvers," in *Proc. FMCAD*. IEEE, 2007, pp. 69–76.

[12] A. Cimatti, A. Griggio, and R. Sebastiani, "Efficient interpolant generation in satisfiability modulo theories," in *Proc. TACAS*, ser. LNCS 4963. Springer, 2008, pp. 397–412.

[13] E. M. 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.

[14] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. TACAS*, LNCS 2988. Springer, 2004, pp. 168–176.

[15] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Proc. TACAS*, ser. LNCS 3440. Springer, 2005, pp. 570–574.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *Proc. POPL*. ACM, 2004, pp. 232–244.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. POPL*. ACM, 2002, pp. 58–70.

[18] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *Proc. CAV*, ser. LNCS 4144. Springer, 2006, pp. 424–437.

[19] K. L. McMillan, "Lazy abstraction with interpolants," in *Proc. CAV*, ser. LNCS 4144. Springer, 2006, pp. 123–136.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Proc. CC*, ser. LNCS 2304. Springer, 2002, pp. 213–228.

[21] R. Sebastiani, "Lazy satisfiability modulo theories," *J. Satisfiability, Boolean Modeling and Computation*, vol. 3, 2007.

Verification of Recursive Methods on Tree-like Data Structures

Jyotirmoy Deshmukh
Dept. of Electrical and Computer Engineering,
University of Texas at Austin.
jyotirmoy@cerc.utexas.edu

E. Allen Emerson
Dept. of Computer Science,
University of Texas at Austin.
emerson@cs.utexas.edu

***Abstract*—Programs that manipulate heap-allocated data structures present a formidable challenge for algorithmic verification. Recursive procedures (methods) in such software libraries are used for a large number of tasks ranging from simple traversals to complex structural transformations. Verification of such methods is undecidable in general. Hence, we present a programming language fragment with a syntax similar to that of C for which correctness can be algorithmically checked. For methods written in our fragment, and specifications in the form of tree automata, verification is efficient in most cases, as illustrated by our prototype tool. Our framework can be used to verify methods such as insertion and deletion of nodes in k-ary trees, binary search trees, linked lists, linked list reversal, and rotations in balanced trees, with respect to specifications such as acyclicity, sortedness, list-ness, tree-ness, and absence of null pointer dereferences.**

I. Introduction

Methods that manipulate heap-allocated, linked data structures such as linked lists, queues, binary search trees and balanced trees are important components of software. Exhaustive verification of such methods requires reasoning about a potentially infinite-state system, which is intractable with conventional model checking. Formally, given a method $\mathcal{M}$, and specifications in the form of pre/post-conditions φ and ψ, *correctness* of $\mathcal{M}$ implies that if each input data structure of $\mathcal{M}$ satisfies φ, then the corresponding output data structure satisfies ψ. Though push-button verification of such methods is highly desirable, there are theoretical limits as to what can be achieved. Methods that manipulate common data structures are often written as recursive programs, and checking correctness of arbitrary recursive methods operating on unbounded data structures is an undecidable problem. However, in this paper we show how certain practical recursive methods can be algorithmically verified.

The underlying framework for verification draws on previous work in [1], which presents an automata theoretic framework for verifying iterative methods that manipulate directed graphs. The main steps in such an approach involve: a) specification of data structure invariants and pre/post-conditions, i.e.*properties* of data structures, using tree automata (cf. [1–4]), b) construction of an *exact* abstraction for a method $\mathcal{M}$ that we term as the *method automaton* $\mathcal{A}_{\mathcal{M}}$, and c) emptiness checking for the product of $\mathcal{A}_{\mathcal{M}}$ with the input property automaton and a negation of the output property automaton. Essentially, correctness of $\mathcal{M}$ is reduced to checking emptiness of the product. In this paper, we use a similar automata-theoretic approach.

The key contributions in this paper are as follows: Compared to previous work that dealt with iterative, non-recursive, methods on general graphs, in this paper, we focus on *recursive methods* that manipulate trees. Secondly, we present a syntactic fragment for such recursive methods, and algorithms to compile methods in this fragment to method automata. The syntax of our fragment is similar to that of C, and a method in our fragment can be compiled using any standard C compiler, after a small amount of syntactic sugaring. Thirdly, we provide mathematical conditions that ensure that every method in our fragment can always be compiled into a meaningful method automaton, and can thus be verified. This is in stark contrast with prior work that assumes an oracle to provide auxiliary proofs about a method's behavior for verification to succeed. For instance, the technique in [1] relies upon a proof that a given method performs only a bounded number of destructive updates to the underlying data structure; in this paper, our syntactic fragment obviates this need. Finally, our techniques can enable verification of practical methods including depth-first insertion and deletion of nodes, search and replace of data in k-ary and binary search trees, linked-list reversal, and tree rotations. Early results with our prototype tool indicate good run-times in verifying useful methods on linked lists and binary trees. Optimizations such as symbolic representations for the automata constructed by our technique would make our approach highly scalable.

In a certain sense, our core methodology can be viewed as an extension of conventional model checking. The most attractive aspect of model checking, i.e., fully automatic verification, *given a set of specifications*, is preserved by our approach. One of the main criticisms of an automata-based approach is that its success hinges on the specification of sufficiently detailed pre/post-conditions. However, a large number of pre/post-conditions are *shape properties*, and are typically generic, i.e., independent of the specific data structure implementation, operating platform, or data values. While we do not focus on specifications in this paper, we argue that by providing such specifications pre-encoded as tree automata (or any suitable logic that can be translated to tree automata), the burden of writing good specifications is lessened. Hence, similar

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

to [1–4], we use tree automata as a specification language. Examples include: acyclicity, sharing, list-ness, sortedness, absence of null pointer dereferences, absence of dangling pointers, and absence of double deletion. The key advantage of our technique is that it does not require annotations such as loop invariants or inductive invariants, and thereby minimizes human input.

The paper is laid out as follows: We introduce the required notation in Sec. II. We discuss the verification framework using automata in Sec. III. In Sec. IV, we provide the syntactic class of recursive methods for which verification is decidable. Experimental results are discussed in Sec. V, and we conclude with related and future work in Sec. VI.

II. Preliminaries

In the programs that we wish to verify, we assume that all program variables belong to only one of two types: a variable over some finite set $\mathcal{D}$ (referred to as a $\mathcal{D}$-variable), and a *pointer variable*. $\mathcal{D}$ is also termed as the data domain. Typical examples of $\mathcal{D}$ include: set of alphanumeric characters, set of integers up to a particular byte-width, set of strings up to a constant length over a finite alphabet, etc. To define a pointer variable, we first introduce the memory model.

Memory Model: A *heap* H is an array of memory cells, where each memory cell contains the value of some program variable. The index of a memory cell is a positive integer known as the *address* of that cell. A *pointer variable* (or simply pointer) p is a variable that evaluates to an address or to 0 (`null`). A *node* n consists of contiguous memory locations that contain the tuple $(d, l_1, \ldots, l_K)$. Here, $d \in \mathcal{D}$, and each l_i is some address. We use $addr(n)$ to denote the address of the first memory location within n. We use $n.d$ (resp. $n.l_i$) to denote the value d (resp. pointer l_i) contained in n. The value $n.d$ is also called the *data value* of n, and $n.l_i$ is also referred to as a *link* of n. We say that a pointer p *points to* n if $p = addr(n)$. If $p = addr(n)$, then the expression $deref(p)$, also called *dereference* of p evaluates to n. The expression $deref(p)$ is undefined and generates an error known as a *null pointer dereference* if the value of p is `null`. We assume that all pointers p are either `null`, or that $deref(p)$ evaluates to a valid node, i.e. we do not allow pointer variables to be explicitly assigned integers, or any form of pointer arithmetic. We assume that the number of links K, is fixed and is part of the definition for a node.

Data Structure: A *data structure* is defined as an arbitrary set of nodes. In this paper, we wish to focus on *rooted* and *connected* tree-like data structures. Such a data structure (say D) has a one-to-one correspondence with a rooted, labeled, directed tree[1] $T(V, E, \mathcal{L}_V, \mathcal{L}_E)$. Here, V is the set of vertices, E is the set of edges, $\mathcal{L}_V : V \to \mathcal{D}$ is a function that maps each vertex to some data value in $\mathcal{D}$, and $\mathcal{L}_E : E \to \{1, \ldots, K\}$ is a function that maps each out-going edge of a given node to a unique positive integer. Each node n in D corresponds to a vertex v_n, such that $\mathcal{L}_V(v_n) = n.d$, and for every n', s.t. $n.l_i = addr(n')$, the edge $e : (v_n, v_{n'})$ belongs to E, and $\mathcal{L}_E(e) = i$. We call every v' s.t. $(v, v') \in E$, a *successor* of v, and analogously call v a predecessor of v'. We assume that there is a designated unique *root* vertex $r \in V$ (corresponding to the root node of the data structure) with the property that r has no predecessor.

[1]A tree is defined in standard fashion as a rooted directed graph in which each node except the root has exactly one predecessor, and the root node has no predecessor.

Methods: A *method* $\mathcal{M}$ is a procedure with an associated signature $sig(\mathcal{M})$ that defines its inputs and return values. For simplicity, we assume that the methods have a *void* return value, i.e. methods do not return anything. Thus, $sig(\mathcal{M})$ effectively defines inputs to $\mathcal{M}$, that are: a) a finite list of pointers $p_1, \ldots, p_n$, and b) a finite list of values $x_1, \ldots, x_m \in \mathcal{D}$. The body of $\mathcal{M}$ may define a finite number of *local* pointer variables and $\mathcal{D}$-variables. $\mathcal{M}$ may additionally access any number of *global* pointer variables and $\mathcal{D}$-variables. A *recursive method* is a method that calls itself. In this paper, we focus on recursive methods that use depth-first traversal to manipulate tree-like data structures. Such a $\mathcal{M}$ satisfies the property that: it is invoked with a pointer to the root r of the tree T as an argument, and for every node n in T, before returning back to the predecessor of n, $\mathcal{M}$ is invoked on all nodes within the sub-tree rooted at n. $\mathcal{M}$ terminates once it returns from recursive invocations on all successors of r.

Def. 1 (Correctness of $\mathcal{M}$)**.** For every input tree T_i that satisfies the pre-condition φ for $\mathcal{M}$, the tree T_o resulting from the action of $\mathcal{M}$ satisfies ψ.

A procedure that can decide the truth or falsehood of the above statement can decide the correctness of $\mathcal{M}$. Unfortunately, this is an undecidable problem. However, in this paper, we argue that by restricting the programming language we can obtain efficiently decidable fragments.

Tree Automata Basics: A K-ary tree has the property that the maximum out-degree of any node in the tree is K. A tree automaton $\mathcal{A}$ running over an infinite K-ary tree T (for some fixed K) is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, \Phi)$ where Σ is a finite, nonempty input alphabet labeling the nodes of T, Q is a finite, nonempty set of states, $\delta : Q \times \Sigma \to 2^{Q^K}$ is the nondeterministic transition function, $q_0 \in Q$ is the initial state, and Φ describes a specific acceptance condition [5]. The run ρ of $\mathcal{A}$ on a Σ-labeled T is an annotation of T with the states Q compatible with δ. The acceptance of T by $\mathcal{A}$ is defined as a specific property of ρ, as described by Φ. For instance, the Büchi condition specifies a set of states (say Q_a) and requires that some state in Q_a appear along every path in ρ infinitely often. The language of $\mathcal{A}$ (denoted $\mathcal{L}(\mathcal{A})$ is the set of all trees accepted by $\mathcal{A}$. Given two automata $\mathcal{A}_1 = (\Sigma_1, Q_1, \delta_2, q_{01}, \Phi_1)$ and $\mathcal{A}_2 = (\Sigma_2, Q_2, \delta_2, q_{02}, \Phi_2)$, the synchronous product $A_p = \mathcal{A}_1 \otimes \mathcal{A}_2$ is defined if $\Sigma_1 = \Sigma_2$. The components of A_p are defined in in terms of $\mathcal{A}_1$ and $\mathcal{A}_2$ as follows: $\Sigma_p = \Sigma_1 = \Sigma_2$, $Q_p = Q_1 \times Q_2$,

$q_{0_p} = (q_{01}, q_{02})$, $\Phi_p = \Phi_1 \times \Phi_2$. Further, let $q_p = (q_1, q_2)$ be a state of $\mathcal{A}_p$. We define $\delta_p(q_p, \sigma)$ as the Cartesian product of each next-state tuple in $\delta_1(q_1, \sigma)$ and $\delta_2(q_2, \sigma)$.

III. Verification using Automata

In this section, we formally define a *method automaton* $\mathcal{A}_\mathcal{M}$ to serve as an exact abstraction for a given method $\mathcal{M}$. We denote the maximum out-degree of any node in a tree T by K (also termed branching-arity). We recursively define a ranking function rk that maps each vertex of a tree T to a unique string from $\{1, \ldots, K\}^*$, as follows: $rk(root)$ = 1 and $rk(n.l_j) = rk(n) \bullet j$, where $\bullet$ denotes string concatenation. Thus, $rk(root.l_2)$ = 12, $rk(root.l_2.l_1)$ = 121, and so on. We define a composite tree T_c as the *superposition* of trees T_i and T_o, and denote the superposition operation as $T_c = T_i \circ T_o$. Essentially, a vertex v_c in T_c is a pair of vertices (v, v'), s.t. $v \in T_i$, $v' \in T_o$, and $rk(v_c) = rk(v) = rk(v')$. For any v in T_i if there is no v' such that $rk(v)$ = $rk(v')$, we let $v' = \perp$ (and similarly for a $v' \in T_o$ corresponding to a missing $v \in T_i$). We define the method automaton $\mathcal{A}_\mathcal{M}$ to run on T_c and accept it if the components of T_c, i.e., T_i and T_o are valid input/output trees for $\mathcal{M}$.

Def. 2 (Mimics)**.** We say that $\mathcal{A}_\mathcal{M}$ *mimics* a method $\mathcal{M}$ (denoted $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$), if for all input trees T_i, $\mathcal{A}_\mathcal{M}$ accepts (T_i, T_o) if and only if $T_o = \mathcal{M}(T_i)$.

Recall from Sec. I that our verification technique combines such $\mathcal{A}_\mathcal{M}$ with the (finite-state) automata for the pre/post-conditions ($\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\psi}$) and checks the product $\mathcal{A}_p$ for non-emptiness. For verification to be solvable, we require that: (a) each of $\mathcal{A}_\mathcal{M}$, $\mathcal{A}_\varphi$, and $\mathcal{A}_{\neg\psi}$ have a decidable emptiness problem, (b) the product operator $\otimes$ for combining these automata is well-defined, and (c) the resulting product $\mathcal{A}_p$ has decidable non-emptiness. In this paper, we restrict our attention to properties specifiable as finite state tree automata; examples can be found in [1]. $\mathcal{A}_\varphi$ and $\mathcal{A}_{\neg\psi}$ thus trivially satisfy the requirement in (a) and (b). However, an arbitrary recursive method $\mathcal{M}$ using the underlying data structure like a tape, is able to read, write, and move on this tape in either direction, and is able to test if a tape location was previously visited. Machines that have these capabilities are, in general, as powerful as Turing machines. However, Turing machines are not useful abstractions for recursive methods, as most interesting problems for them are undecidable. To overcome this problem, in Lem. 1, we show that if a method $\mathcal{M}$ performs only a "bounded amount of work" on its input data structure, then there exists a finite state tree automaton $\mathcal{A}_\mathcal{M}$, s.t. $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$. Such an $\mathcal{A}_\mathcal{M}$ is of value, as it clearly satisfies the requirements in (a), (b) and (c). To precisely explain the notion of bounded work, we first define a *destructive update*.

Def. 3. A *destructive update* (du) is a modification to a data structure node n (directly by accessing n or indirectly through a pointer p to n). Let x be some element of $\mathcal{D}$, and let p_i's be pointers, then du has one of the following forms:

- $n.d = x$ or $p\text{->}d = x$ (changing the data value of n),
- $n.l_i = p_j$ or $p\text{->}l_i = p_j$ (changing a link of n),
- $delete(p)$ (marking the node pointed to by p as free),
- $p\text{->}l_j = new(x, p_1, \ldots, p_K)$ (inserting a new node as a successor of $deref(p)$).

Lemma 1. *Let the maximum number of destructive updates performed by $\mathcal{M}$ on any node of the input data structure D_i, before it terminates, be r. If $r \leq c$ for some constant c, then there exists a finite state automaton $\mathcal{A}_\mathcal{M}$, such that $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$.*

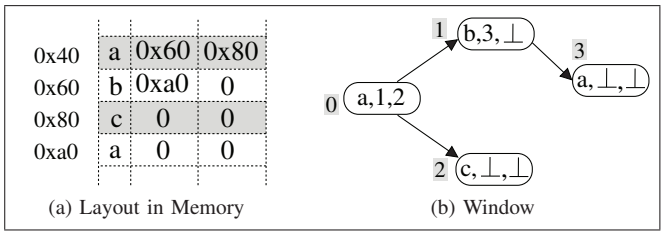
Proof: For simplicity, consider the case where $r = 1$. Let $T_c = T_i \circ T_o$. Each vertex n_c in T_c is the pair (n_i, n_o). We can now define $\mathcal{A}_\mathcal{M}$ to run on T_c, starting at the root of T_c in state q_0. We can view the single destructive update performed by $\mathcal{M}$ on n_i as a function f_{du} that maps each n_i to some output node n_o. $\mathcal{A}_\mathcal{M}$ transitions to a reject state rej if $n_o \neq f_{du}(n_i)$, otherwise it remains in the state q_0 along all successors of n_i. Finally, if $\mathcal{A}_\mathcal{M}$ reaches a terminal vertex (i.e., no successors) in T_c, it transitions to acc, and stays in that state. If $\mathcal{A}_\mathcal{M}$ accepts T_c (i.e., reaches a terminal node along every path in T_c) then it must be true that each (n_i, n_o) encodes a valid action of $\mathcal{M}$. In other words, if $T_c = T_i \circ T_o$ is accepted by $\mathcal{A}_\mathcal{M}$, then $T_o = \mathcal{M}(T_i)$. If $r > 1$, then we can define T_c = $T^0 \circ T^1 \ldots \circ T^r$, where $T^0 = T_i$ and $T^r = T_o$, and $\mathcal{A}_\mathcal{M}$ accepts T_c if for each pair (T^j, T^{j+1}), $T^{j+1} = f_{du_j}(T^j)$. By construction, if r is a constant, we can define $\mathcal{A}_\mathcal{M}$ that can mimic $\mathcal{M}$ using only a finite input alphabet that is proportional to the arity of the tree K and r, and uses only a finite number of states. Thus, if $\mathcal{M}$ performs only r ($\leq$ some c) destructive updates to each node in T_i, there always exists a finite $\mathcal{A}_\mathcal{M}$ such that $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$. ■

We refer to Lem. 1 as the *bounded updates property*. Unfortunately, we can show that it is impossible to determine whether an arbitrary recursive method satisfies the bounded updates property. The proof is based on a reduction from Rice's theorem [6], and we skip it for brevity. While Lem. 1 shows that for any $\mathcal{M}$ that has the bounded updates property, there is *some* $\mathcal{A}_\mathcal{M}$ s.t. $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$, it does not provide a recipe for extracting $\mathcal{A}_\mathcal{M}$ from $\mathcal{M}$. Also, trying to compile an $\mathcal{A}_\mathcal{M}$ from an arbitrary $\mathcal{M}$ is futile, due to the undecidability barrier. In this paper, we identify a syntactic class of methods obtained by restricting the recursive calling patterns and the pointer usage by these methods, such that for any method $\mathcal{M}$ belonging to this fragment, we can automatically obtain a finite state tree automaton $\mathcal{A}_\mathcal{M}$. Before we inspect this class, we introduce window-based abstraction, which helps in defining the input alphabet for method automata.

Window-based Abstraction: A window w is a finite encoding of a node n and a bounded number of nodes that succeed n, similar to the finite encodings developed in [1, 2]. A window is obtained by mapping arbitrary integer addresses to a small, finite subset of integers. For the window-based abstraction to be applicable, we need the following assumptions: 1) $\mathcal{M}$ does not access global pointer variables, and, 2) $sig(\mathcal{M})$ has exactly one pointer argument denoted by `I` (short for iterator), and a finite number of $\mathcal{D}$-valued arguments. Let $v_\mathtt{I}$ denote the node

pointed to by I, i.e., $v_{\mathtt{I}} = deref(\mathtt{I})$. In a given invocation of $\mathcal{M}$, since I is the only pointer passed as an input to $\mathcal{M}$, other nodes have to be accessed by following the links of $v_{\mathtt{I}}$. In any reasonable syntax, this limits $\mathcal{M}$ to accessing a small set of nodes. Borrowing notation from C, we use $\mathtt{I}{->}l_i$ to denote $v_{\mathtt{I}}.l_i$. Similarly, $\mathtt{I}{->}l_i{->}l_j$ denotes $deref(v_{\mathtt{I}}.l_i).l_j$. Each expression of the form $\mathtt{I}{->}l_i{->}l_j\ldots$ is called a *pointer expression*. Any method body contains a finite number of such pointer expressions, which can be statically identified. Let $PE_{\mathcal{M}}$ denote this set.

Def. 4 (Window)**.** A *window* $w(v_{\mathtt{I}})$ is a tuple = $\{v_{\mathtt{I}}, u_1, \ldots, u_n\}$, where each u_i is obtained by dereferencing an element of $PE_{\mathcal{M}}$. I.e., each u_i has the property that if $u_i = deref(p)$ for some $p \in PE_{\mathcal{M}}$, then $(v_{\mathtt{I}}, u)$ is an element of one of : E, E^2, …, E^z, for a finite z.



(a) Layout in Memory (b) Window

Fig. 1: Window-based Abstraction

Informally, each u_i is a node that is some descendant of $v_{\mathtt{I}}$. Given a set of concrete nodes, we define a function $laddr$ that maps each u_i to i, and $v_{\mathtt{I}}$ to 0. The local address function $laddr$ thus replaces each actual address with an "abstract address". For a node in the concrete data structure that has a successor that lies outside the window, the corresponding link is marked with a '$*$', and null pointers are marked with '$\perp$'. Fig. 1a shows the layout of four nodes of a binary tree in the memory. A window of size 4 representing this heap structure is shown in Fig. 1b, where the numbers above each node indicate the $laddr$ of that node.

By encoding an input tree T using windows, we can obtain an equivalent tree T_w, s.t., every node $v_{\mathtt{I}}$ in T maps to the vertex $w(v_{\mathtt{I}})$ in T_w. In other words, every vertex of T_w is a window; T is embedded within T_w, and can be extracted from T_w by eliding all nodes, except the first, from each vertex in T_w. We note that an arbitrary tree T'_w in which each vertex is a window may not correspond to a valid tree. Since windows encode sets of nodes in T, adjacent windows in T'_w may contain different values for the same nodes in T. We say that T_w is a *consistent tree*, if the overlapping portions of adjacent windows in T_w are identical (except for the values marked with $*$), and for the window corresponding to a terminal node $v_{\mathtt{I}}$, all other nodes in the window are marked as null ($\perp$).

Example 1. Consider a node n of the form: $n = (d, l_1)$. Fig. 2 illustrates a list T_w in which every vertex is a window of size 2, and T_w is consistent with T.

Modeling destructive updates: A destructive update du to $v_{\mathtt{I}}$, either marks $v_{\mathtt{I}}$ as deleted, or changes $\mathtt{I}{->}d$ or $\mathtt{I}{->}l_i$ (for some i). We can view du as a function mapping an "input" window w_i to an "output" window w_o, where w_o is obtained by performing the actions of du on w_i. Thus for statement: $\mathtt{I}{->}d = x$; w_o is identical to w_i except for $\mathtt{I}{->}d$, which has the value x. When du modifies $\mathtt{I}{->}l_i$, the expression on the RHS of du is a pointer expression, or a new node. The effect of statement $\mathtt{I}{->}l_i := p$, is to set $\mathtt{I}{->}l_i$ in w_o to $laddr(p)$. Insertion of a new node is modeled by adding a new node to w_o and setting $\mathtt{I}{->}l_i$ to the $laddr$ of this new node. Deleting a node is modeled by over-writing each field in the corresponding node in w_o with some special character (say '$-$'). Thus, for any destructive update du, we can compute a function f_{du} that maps a given w_i to some w_o.

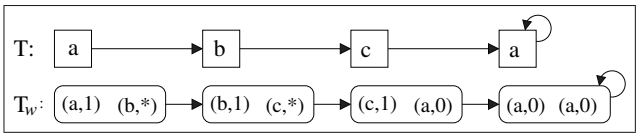


Fig. 2: Consistent Abstraction

Example 2. Consider a node with the same layout as in Ex. 1. Let $w(v_{\mathtt{I}}) = \{(a,1),(b,*)\}$, where $v_{\mathtt{I}} = (a,1)$. Now consider the action of the destructive update du: $\mathtt{I}{->}d := \mathtt{I}{->}l_1{->}d$. The resulting window $w'(v_{\mathtt{I}})$ is $\{(b,1),(b,*)\}$. In this case, f_{du} is a function mapping any window w to a window w' in which both nodes in w' have the same data value as the data value of the second node in w.

IV. Syntactic Class for Recursive Methods

In this section, we present a syntactic class for depth-first recursive methods on tree-like data structures for which verification is efficiently decidable. We split this class into the class of tail-recursive methods, and the more general class allowing non-tail recursion. Tail recursive methods can be converted into iterative methods, and one could argue that these can then be verified using techniques developed in [1]. However, the purpose of discussing tail-recursive methods is twofold: a) it displays how the syntax that we present ensures the bounded updates property, which contrasts with [1], where the bounded updates property had to be ensured by an oracle, and b) it makes the exposition on the more general class easier by introducing necessary terminology in the context of an easier problem.

We say that a method $\mathcal{M}$ visits a node $v_{\mathtt{I}}$ when it is invoked with I as its pointer argument. The control-flow structure of a general recursive method $\mathcal{M}$ is as follows: $\mathcal{M}$ first visits the *root* node. Any other node $v_{\mathtt{I}}$ is visited following a recursive invocation from $v_{\mathtt{I}}$'s parent. $\mathcal{M}$ re-visits $v_{\mathtt{I}}$ interleaved with recursive invocations with successors of $v_{\mathtt{I}}$ as arguments, before it finally *returns* back to the parent of $v_{\mathtt{I}}$ from which it was invoked. In a *tail recursive* method all recursive calls are the final operations in the method body, before it returns.

Tail-Recursive Methods: We formally present the syntax for tail-recursive methods in Fig. 3. Each $\mathcal{M}$ has a signature

sig and a body. $\mathcal{M}$'s body is sub-divided into base case blocks ($baseblks$), and a recursive block. The semantics are as follows: $\mathcal{M}$ first evaluates $bcond_j$ over the nodes in $w(v_{\mathtt{I}})$, and if *true*, performs the actions within $bblock_j$ followed by a `return`. If none of the $bcond_j$ evaluate to *true*, then $\mathcal{M}$ performs the destructive updates specified by $rblock_0$ over the nodes within $w(v_{\mathtt{I}})$, followed by recursive visits to successors of $v_{\mathtt{I}}$ for which $rcond_j$ evaluates to *true* over the possibly updated $w(v_{\mathtt{I}})$. Block statements ($block$) are recursively defined to consist of conditional statements ($ifstmt$), local assignments ($local$), destructive update statements (du), or empty statements ($skip$). We omit the syntax for assignments to local variables ($local$), and the syntax for du statements is specified in Def. 3. The expressions exp, $bcond_j$ and $rcond_j$ are Boolean-valued equality or disequality expressions comparing: a) two pointers, b) a pointer with the null value ($\perp$), or c) data variables. We assume that any calls to methods other than $\mathcal{M}$ are inlined within the body of $\mathcal{M}$. In contrast to standard programming languages, we specifically disallow loops and pointer arithmetic.

$\mathcal{M}$	::=	sig { $baseblks$ $rblock_0$ $calls$ `return` }
sig	::=	(`I`) | (`I`, $x_1, \ldots, x_n$)
$baseblks$	::=	$baseblk$ | $baseblks$ $baseblk$
$baseblk$	::=	`if` ($bcond_j$) { $bblock_j$ `return`}
$\forall j : bblock_j$	::=	$block$
$rblock_0$	::=	$block$
$calls$	::=	$call$ | $calls$ $call$
$call$	::=	`if` ($rcond_j$) $\mathcal{M}$(`I->`l_j)[a]
$block$	::=	$stmt$ | $block$ $stmt$
$stmt$	::=	$ifstmt$ | du | `skip` | $local$
$ifstmt$	::=	`if` (exp) { $block$ } `else` { $block$ }

[a]Note: We statically enforce that for any two invocations of the form $\mathcal{M}$(`I->`l_j) and $\mathcal{M}$(`I->`l_k), `I->`$l_j \neq$ `I->`l_k, failing which is a syntax error.

Fig. 3: Syntax for Tail-Recursive methods

Lemma 2. *Methods that respect the syntax specified in Fig. 3 satisfy the bounded updates property.*

The correctness of Lem. 2 follows from the observation that our syntax ensures that $\mathcal{M}$ recursively visits each successor of any node $v_{\mathtt{I}}$ at most once[2] - when it is recursively invoked from the parent of $v_{\mathtt{I}}$. Moreover, as no destructive update is performed when $\mathcal{M}$ returns, effectively, $\mathcal{M}$ destructively updates any $v_{\mathtt{I}}$ at most once. From Lem. 1, we know that there exists some finite state automaton $\mathcal{A}_{\mathcal{M}}$ such that $\mathcal{A}_{\mathcal{M}} \bowtie \mathcal{M}$. We show how we can derive the desired $\mathcal{A}_{\mathcal{M}}$ in Algo. 1.

Recall that $\mathcal{A}_{\mathcal{M}}$ is defined to run on a composite tree T_c, where the set of nodes of T_c (i.e. the input alphabet for $\mathcal{A}_{\mathcal{M}}$) is a set of pairs of windows, i.e., $\Sigma = \{\sigma | \sigma = (w_{in}, w_{out})\}$. The required size and form of each of the windows can be statically computed in a single pass over $\mathcal{M}$ by inspecting the set $PE_{\mathcal{M}}$ (pointer expressions within $\mathcal{M}$). Note that our syntax

[2]Of special note is the case where $\mathcal{M}$ visits the l_1-successor of $v_{\mathtt{I}}$, followed by updating $v_{\mathtt{I}}$ so that $v_{\mathtt{I}}.l_2 = v_{\mathtt{I}}.l_1$, followed by visiting $v_{\mathtt{I}}.l_2$ (which is now the same as $v_{\mathtt{I}}.l_1$). Such a scenario is detected at compile-time and disallowed, as specified by the footnote in Fig. 3.

Algorithm 1: CompileTailRecursive

```
1  begin
2      W:= all windows (computed by inspecting PE_M),
       Σ:=W × W, q_0:=init, Q:={init, acc, rej} ∪ Q_Σ,
       Φ:={acc}, H :={}
       /* Note that σ = (w_in, w_out) */
3      foreach σ in Σ, q in Q_Σ do
           /* Reject inconsistent (q,σ) */
4          H :=H ∪ {(σ, init)}
5          if (consistent(σ,q)) then H :=H ∪ (σ, q)
6          else reject(q,σ)
7      RS:=Σ
8      foreach baseblk in baseblks, σ in Σ do
9          if (σ ⊨ bcond_j) then
10             RS :=RS \ {σ}
11             f_bblock_j := computeBlock(bblock_j)
12             if (w_out == f_bblock_j(w_in)) then
                   /* σ mimics bblock_j */
13                 foreach q in H(σ) do accept(q,σ)
14             else
                   /* σ does not mimic bblock_j */
15                 foreach q in H(σ) do reject(q,σ)
16     foreach σ in RS do
17         f_rblock_0 :=computeBlock(rblock_0)
18         if (w_out == f_rblock_0(w_in)) then
               /* σ mimics rblock_0 */
19             foreach q in H(σ) do
20                 foreach j in {1, ..., K} do
21                     if (σ ⊨ rcond_j) then
22                         next[j]:= computeState(σ)
23                     else
24                         next[j]:=acc
25                 δ:= δ ∪ {(q, σ, next)}
26         else
               /* σ does not mimic rblock_0 */
27             foreach q in H(σ) do reject(q,σ)
28 end
```

guarantees that the size and number of possible windows is finite, and bounded by the largest expression in $PE_{\mathcal{M}}$; thus, the size of Σ (which is simply the set of all pairs of windows) is finite. The set of states $Q = \{acc, rej, init\}$ have their usual meanings as an *accept* state, a *reject* state, and an *initial* state respectively. The states in Q_Σ are used to check if $w(v_{\mathtt{I}})$ is consistent with the window(s) read at predecessors of $v_{\mathtt{I}}$. If $|w(v_{\mathtt{I}})| = 1$, then Q_Σ is empty, and then Lines 2-6 are skipped. Otherwise, we reject those state/symbol pairs that correspond to an inconsistent annotation at neighboring nodes in T_c. As Σ is finite, the set of states Q is also finite. If a state/symbol pair is consistent, we add it to the map H in Line 5.

We then identify those symbols σ that correspond to $\mathcal{M}$ entering any of the base cases. For the $bblock_j$ block of statements within the j^{th} base case, we compute f_{bblock_j} that is the composition of the functions for the statements within $bblock_j$. If the update encoded by σ is faithful to f_{bblock_j}, i.e. $w_{out} = f_{bblock_j}(w_{in})$, we accept all consistent states for

this symbol (Line 13), else we reject them (Line 15). Once all $baseblk$ statements are processed, the remaining symbols (in the set RS) correspond to symbols for which $\mathcal{M}$ enters the recursive case. For each symbol $\sigma \in RS$, we check if the update encoded by σ is faithful to f_{rblock_0}. If not, we reject all consistent states for that σ (Line 27). If yes, we identify the successors of $v_{\mathtt{I}}$ within $w_{out}(v_{\mathtt{I}})$ that $\mathcal{M}$ would visit (by virtue of $rcond_j$ evaluating to $true$), and transition to the appropriate state (in Q_Σ) for those successors (Line 22). The remaining successors are not visited by $\mathcal{M}$, and hence, we simply transition to acc for these (Line 24). We use `reject` (q,σ) as a macro that adds the transition: $(q, \sigma, (rej, \ldots, rej))$ to δ (similarly `accept`). The function `computeBlock` composes the functions f_s for individual statements s within a block statement. `getState` returns the state in Q_Σ that encodes the value of the current pair of windows (σ). `consistent` checks if a given $q \in Q_\Sigma$ that encodes the values in some preceding window, is consistent with the window being currently processed. As a final step (not shown in the algorithm), we add self-loops to the acc and rej states, making them "trap" states. Note that by construction, our algorithm guarantees that Lem. 3 is true. The description presented above gives a proof sketch; we omit the details for brevity.

Lemma 3. *$\mathcal{A}_\mathcal{M}$ derived using Algo. 1 has the following properties: a) $\mathcal{A}_\mathcal{M}$ is a finite tree automaton, i.e., a finite Q and Σ, b) if $\mathcal{A}_\mathcal{M}$ accepts a composite tree $T_c = T_i \circ T_o$, then $T_o = \mathcal{M}(T_i)$, i.e., $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$.*

General Recursive Methods: In contrast to tail-recursive methods, a non-tail-recursive method can re-visit a node between recursive calls to its successors, and perform destructive updates. We make the same assumptions as for tail-recursive methods that: a) methods do not use global pointers, b) methods use a single pointer argument during recursive invocation, and c) at any node, for any given successor s, $\mathcal{M}$ is invoked with s as an argument at most once. The syntax for the general class of recursive methods is presented in Fig. 4. Instead of a single destructive update block $rblock_0$ as in Fig. 3, we allow up to K $cblock_j$ statements, where each $cblock_j$ statement consists of a recursive call to the j^{th} successor of $v_{\mathtt{I}}$, followed by a destructive update block $rblock_j$. We only show the differing parts in this description, as all other definitions remain the same (except $calls$/$call$, which is no longer relevant).

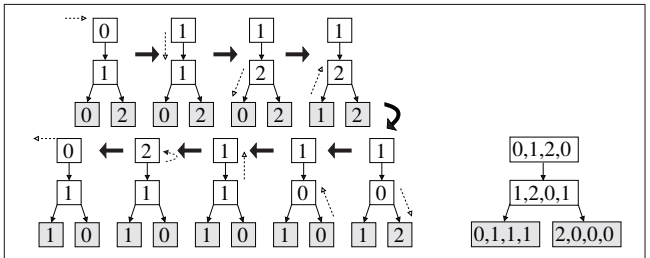
$\mathcal{M}$	::=	sig { $baseblks$ $rblock_0$ $cblocks$ `return` }
$cblocks$	::=	$cblock_j$ | $cblocks$ $cblock_j$
$cblock_j$	::=	$call_j$ $rblock_j$
$rblock_j$	::=	$block$

Fig. 4: Syntactic Class for Recursive Methods

Lemma 4. *Methods with syntax as specified by Fig. 4 satisfy the bounded updates property.*

Proof: A recursive method $\mathcal{M}$ satisfying the above assumptions visits a node with K out-going edges and performs destructive updates: a) the first time when called from the parent node of n ($rblock_0$), and b) K times after each recursive call returns ($rblock_{1..K}$), and c) never after the return. Thus, the total number of destructive updates to any node is at most $K + 1$. However, for a given tree, K is a constant, and thus the total number of destructive updates is bounded by a constant. Thus, such an $\mathcal{M}$ satisfies the bounded updates property. ■

The overall scheme for compilation into $\mathcal{A}_\mathcal{M}$ for the class in Fig. 3 is similar to the one used for compiling tail-recursive methods. In Algo. 1, all destructive updates by a tail-recursive method on a given window can be composed into a single destructive update described by the function f_{rblock_0} using `computeBlock`. This is not possible for a method belonging to Fig. 4, since it performs $K + 1$ distinct blocks of updates. However, by altering the way we define the composite tree, we can use an algorithm very similar to Algo. 1 to compile $\mathcal{M}$ into $\mathcal{A}_\mathcal{M}$. We first observe that if we record the actions of such a $\mathcal{M}$ during a depth-first traversal at each node in the underlying data structure D, we obtain an annotated D', where every node of D' is a $K + 2$-tuple of values. We illustrate this with an example in Ex. 3.



(a) Depth-first Traversal (b) Annotated Tree

```
method changeData (I) {
    if ((I->l1 == ⊥)&&(I->l2 == ⊥)) {
        incMod3(I->d);
        return;
    }
    incMod3 (I->d);
    if (I->l1 ≠ ⊥) { changeData (I->l1); }
    incMod3 (I->d);
    if (I->l2 ≠ ⊥) { changeData (I->l2); }
    incMod3 (I->d);
    return;
}
```

(c) Method `changeData`

Fig. 5: Depth-first Traversal by `ChangeData`

Example 3. Fig. 5c shows the recursive method `changeData`, that changes the data value of each node in the input tree. We assume that $\mathcal{D}$ is the set $\{0, 1, 2\}$, and use `incMod3` as a macro to replace the three conditional assignments that specify $modulo$-3 increment. Fig. 5a shows the actions of `changeData` on an input tree, while Fig. 5b shows the input tree annotated with the actions of `changeData`.

The intuition for Algo. 2 is that depth-first traversal by a method $\mathcal{M}$ generates an annotation similar to that in Ex. 3 on the underlying tree. We define the composite tree T_c s.t., each node σ in T_c is a $(K + 2)$-tuple of the form

Algorithm 2: CompileGeneralRecursive

```
1  begin
       /* Note that
          σ = (w_in = w_0, w_1, ..., w_K, w_out = w_{K+1})  */
2      Lines 2-15 of Algo. 1
3      foreach σ in RS do
4          rejected:=false
5          foreach j in {0, ..., K} do
6              f_{rblock_j} := computeBlock(rblock_j)
7              if (w_{j+1} ≠ f_{rblock_j}(w_j)) then
8                  foreach q in H(σ) do reject(q, σ)
9                  rejected := true
10                 break
11         if (¬rejected) then  Lines 19-25 of Algo. 1
12 end
```

Method	Spec.	Time[a] (secs) $\mathcal{A}_\mathcal{M}$	Time[a] (secs) Total	Mem. (MB)
On Linked Lists:				
`DeleteNode`	Acyclic	0.3	1.3	20
`InsertAtTail`	Acyclic	0.01	0.8	<1
`InsertNode`	Acyclic	0.4	1.6	48
On Binary Trees:				
`InsertNode`	Acyclic	15	329	2512
`ReplaceAll(a,b)`	Acyclic	5	26	324
	$\nexists$`I : I->`$d = a$	5	27	432
`DeleteLeaf`	Acyclic	12	48	630

[a]Experiments were performed on an Athlon 64X2 4200$^+$ system with 6GB RAM.

TABLE I: Experimental Results

$(w_0, w_1, \ldots, w_K, w_{K+1})$. For a composite tree T_c being inspected by $\mathcal{A}_\mathcal{M}$, $w_{in} = w_0$, $w_{out} = w_{K+1}$, and each w_{j+1} is the conjectured value for the effect of $rblock_j$. Our syntax ensures that a recursive call to the children of w_j does not affect w_j. Hence, we can check if for the function f_{rblock_j} corresponding to each $rblock_j$ statement, whether $w_{j+1} = f_{rblock_j}(w_j)$, for all j s.t. $0 \leq j \leq K$ in a single swoop. If any of these conditions is false, we reject the entire symbol σ (Line 8), else we proceed in the same way as in Algo. 1. Due to space limitations, Algo. 2 only shows the parts that differ from Algo. 1. The definitions of `consistent`, `getState` and `computeBlock` are changed to account for the changes in σ.

To make a correctness argument, we introduce some notation. Let T_c be a tree in which every node is a $K+2$-tuple of windows. Let T_{i_w} be the tree obtained by eliding all but the first window in every node of T_c, and T_{o_w} be obtained by eliding all but the last window in every node of T_c. Suppose T_{i_w} and T_{o_w} are consistent trees (recall the definition from Sec. III), then let T_i (resp. T_o) be the corresponding tree for which T_{i_w} (resp. T_{o_w}) is the window-based abstraction. By construction, Algo. 2 guarantees Theorem 1; we skip the proof for brevity.

Theorem 1. *$\mathcal{A}_\mathcal{M}$ derived using Algo. 2 has the following properties: a) $\mathcal{A}_\mathcal{M}$ is a finite tree automaton, i.e., a finite Q and Σ, b) $\mathcal{A}_\mathcal{M}$ rejects T_c if T_{i_w} or T_{o_w} are inconsistent, and c) if $\mathcal{A}_\mathcal{M}$ accepts T_c, then $T_o = \mathcal{M}(T_i)$, i.e. $\mathcal{A}_\mathcal{M} \bowtie \mathcal{M}$.*

V. Experimental Results

The complexity of our technique is equal to the complexity of checking emptiness of the product automaton $\mathcal{A}_p$. For acceptance conditions such as the Büchi condition, this is polynomial in the number of states of $\mathcal{A}_p$, which is itself linear in the size of $\mathcal{A}_\mathcal{M}$ (denoted $|\mathcal{A}_\mathcal{M}|$), $|\mathcal{A}_\varphi|$ and $|\mathcal{A}_{\neg\psi}|$. $|\mathcal{A}_\mathcal{M}|$ is proportional to the size of $\mathcal{M}$, but is dominated by $|\Sigma|$, which in turn is polynomial in $|\mathcal{D}|$ and exponential in the branching-arity (K) of the tree. We note that for purely structural properties, $\mathcal{D}$ can often be abstracted to a single symbol. We have implemented a prototype tool in `Java` that can verify methods with the syntax specified by Fig. 4. Some of the early results are shown in Table I. The methods we used for testing are commonly found implementations for linked lists and binary trees written in `C`, adapted to our syntax. As seen in Table I, while the time required to construct $\mathcal{A}_\mathcal{M}$ is a fraction of the total time taken, the memory consumption is a sore spot. This is so because, in our current implementation, we explicitly construct Σ. The size of Σ can be reduced by several orders of magnitude by an abstraction that involves forming clusters of similar symbols in Σ. Furthermore, we can employ symbolic techniques like BDDs or SAT to compactly represent Σ, which would ameliorate the memory consumption. Lastly, as predicted by the complexity analysis, we observe that the run-time and memory consumption increases sharply with K.

Counterexample Generation: If $\mathcal{A}_p$ is found to be non-empty, the tree T_c witnessing its non-emptiness can be extracted from the transition diagram δ_p of $\mathcal{A}_p$ using standard techniques. By projecting T_c onto its components, we can obtain trees T_i and T_o respectively. T_i represents a valid input tree to $\mathcal{M}$ for which the "bad" output T_o is generated, i.e., a concrete counterexample to the correctness of $\mathcal{M}$.

VI. Related Work and Conclusions

Shape Analysis: Shape analysis [7–9], focuses on computing (3-valued) structure descriptors at each program point, typically using static analysis. Shape analysis can be used to analyze a broad class of methods, but to our best knowledge provides approximate results in double exponential time. Predicate abstraction has been used for shape analysis in [10–12]. [10, 12] focus on singly linked lists, and [13] extends the authors' previous work to programs with single-parent heaps. While [14] provides a way to combine predicate abstraction and model checking, it may require hints to converge to a solution.

Separation Logic: Separation logic, typically allows deductive verification for heap modifying programs [15, 16], and has been traditionally used for manual proofs or in conjunction with a theorem prover. Recent work has focussed on automation, by deriving decidable fragments for programs operating on structures with single successors [17].

Automata-based approaches: This paper significantly extends prior work in [1], which uses tree automata for verifying iterative methods. In [2, 3], system configurations are trees, succinctly encoded as tree automata. The transition relation is a bottom-up tree transducer τ, and the technique checks if τ^* applied to the initial configuration automaton reaches a bad state. Though this is undecidable, the authors use abstraction-refinement to obtain a conservative solution. [4] uses tree automata with size constraints to verify balanced trees implementations. PALE [18] encodes programs and partial specifications in MSO logic, which has a non-elementary decision procedure.

Logic-based Approaches: [19] describes a logic of reachable patterns that is undecidable, which when restricted to certain reachability patterns, yields a decidable fragment that can be checked in double exponential time. The restrictions imposed to obtain decidability are incomparable to the work in this paper. While the work in [20] inspires some of the later work on decidable fragments (cf. [19]), the paper itself does not yield a practical algorithm. Bottom-up shape analysis [21] for heap-manipulating programs computes Hoare triples as summaries for a given method. It may be possible to combine our technique with bottom-up analysis by substituting method fragments that do not respect our imposed syntax rules with equivalent summaries, thereby allowing us to model a larger class of methods.

Due to space restrictions, we omit a few simple extensions that our technique can handle such as: allowing methods to return values in a restricted form (both $\mathcal{D}$-values and pointer values), allowing methods to have a limited access to predecessor nodes up to a bounded distance, and allowing more than one pointer argument in the method signature, with the restriction that all arguments are contained within a window. A more significant extension is verification of recursive methods on directed acyclic graphs (*dags*). Since *dags* can contain sharing between nodes, the restriction of "one visit per successor" is not enough to ensure the bounded updates property. However, if we can enforce (or guarantee) that a method visits each node in a *dag* (and thus every sub-*dag*) at most once, then such methods would satisfy the bounded updates property, and could be verified using a modified form of the algorithms presented here.

Acknowledgements: We thank Prateek Gupta for insightful discussions in the initial phase of this work, and the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

[1] J. Deshmukh, E. Emerson, and P. Gupta, "Automatic verification of parameterized data structures," in *Proc. of TACAS*, 2006, pp. 27–41.

[2] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar, "Verifying programs with dynamic 1-selector-linked structures in regular model checking," in *Proc. of TACAS*, 2005, pp. 13–29.

[3] A. Bouajjani, P. Habermehl, and A. Rogalewicz, "Abstract regular tree model checking of complex dynamic data structures," in *Proc. of SAS*, 2006, pp. 52–70.

[4] P. Habermehl, R. Iosif, and T. Vojnar, "Automata-based verification of programs with tree updates," in *Proc. of TACAS*, 2006, pp. 350–364.

[5] E. A. Emerson and C. S. Jutla, "The complexity of tree automata and logics of programs," *SIAM J. Comput.*, vol. Vol. 29, pp. 132–158, 1999.

[6] M. Sipser, *Introduction to the Theory of Computation*, 1st ed. Course Technology, Dec. 1996.

[7] N. Rinetzky and M. Sagiv, "Interprocedural shape analysis for recursive programs," in *Proc. of CC*, 2001, pp. 133–149.

[8] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Trans. Program. Lang. Syst.*, vol. Vol. 24, pp. 217–298, 2002.

[9] T. Lev-Ami, N. Immerman, and M. Sagiv, "Abstraction for shape analysis with fast and precise transformers," in *Proc. of CAV*, 2006, pp. 547–561.

[10] I. Balaban, A. Pnueli, and L. D. Zuck, "Shape analysis by predicate abstraction," in *Proc. of VMCAI*, 2005, pp. 164–180.

[11] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv, "Predicate abstraction and canonical abstraction for singly-linked lists," in *Proc. of VMCAI*, 2005, pp. 181–198.

[12] J. Bingham and Z. Rakamaric, "A logic and decision procedure for predicate abstraction of heap-manipulating programs," in *Proc. of VMCAI*, 2006, pp. 207–221.

[13] I. Balaban, A. Pnueli, and L. Zuck, "Shape analysis of single-parent heaps," in *Proc. of VMCAI*, 2007, pp. 91–105.

[14] D. Dams and K. Namjoshi, "Shape analysis through predicate abstraction and model checking," in *Proc. of VMCAI*, 2003, pp. 310–323.

[15] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proc. of LICS*, 2002, pp. 55–74.

[16] D. Distefano, P. OHearn, and H. Yang, "A local shape analysis based on separation logic," in *Proc. of TACAS*, 2006, pp. 287–302.

[17] J. Berdine, C. Calcagno, and P. O'Hearn, "A decidable fragment of separation logic," in *Proc. of FSTTCS*, 2004, pp. 97–109.

[18] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proc. of Programming Language Design and Implementation*, 2001, pp. 221–231.

[19] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani, "A logic of reachable patterns in linked data-structures," in *Proc. of FOSSACS*, 2006, pp. 94–110.

[20] M. Benedikt, T. Reps, and M. Sagiv, "A decidable logic for describing linked data structures," in *Proc. of ESOP*, 1999, pp. 2–19.

[21] B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori, "Bottom-up shape analysis," in *Proc. of SAS*, 2009, pp. 188–204.

MCC: A runtime verification tool for MCAPI user applications

Subodh Sharma
School of Computing
University of Utah
Salt Lake City, UT 84112
svs@cs.utah.edu
www.cs.utah.edu/~svs

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112
ganesh@cs.utah.edu
www.cs.utah.edu/~ganesh

Eric Mercer
Computer Science Department
Brigham Young University
Provo, UT 84602.
eric.mercer@byu.edu
http://faculty.cs.byu.edu/~egm

Jim Holt
Networking and Multimedia Group
Freescale Semiconductor, Inc.
7700 West Parmer Lane, MD:PL51
Austin, TX 78749
Jim.Holt@freescale.com

***Abstract*—We present a dynamic verification tool MCC for Multicore Communication API applications – a new API for communication among cores. MCC systematically explores all relevant interleavings of an MCAPI application using a tailor-made dynamic partial order reduction algorithm (DPOR). Our contributions are (i) a way to model the non-overtaking message matching relation underlying MCAPI calls with a high level algorithm to effect DPOR for MCAPI that controls the lower level details so that the intended executions happen at runtime; and (ii) a list of default safety properties that can be utilized in the process of verification. To our knowledge, this is the first push button model checker for MCAPI application writers that, at present, deals with an interesting subset of MCAPI calls. Our result is the demonstration that we can indeed develop a dynamic model checker for MCAPI that can directly control the non-deterministic behavior at runtime that is inherent in any implementation of the library without additional API modifications or additions..**

I. Introduction

Future embedded systems will employ multiple and heterogeneous cores (CPU, DSP, *etc.*) and run a large amount of thread-based shared-memory and message-passing based software. To permit software reuse and derive the benefits of standardization, an API for multicore communication (MCAPI) is being developed by a group of over 25 leading companies [1]. Unlike large existing APIs such as MPI [2] that are meant for the high-end compute clusters, MCAPI is being designed ground-up from a clean slate to address the needs of embedded multicore systems. MCAPI supports connectionless messages, connection-oriented packets, and even scalar (bus based) transfers. The example in Section II-B shows how an MCAPI application might be written using POSIX threads (Pthreads, [3]) for orchestrating the overall computation.

This paper describes the first *dynamic* (or *runtime*) formal verification tool for MCAPI applications called MCC (MCAPI Checker) where dynamic means that the verification process takes place at run time using the MCAPI runtime environment. It is practically impossible to construct verification models or state transition relations that accurately model the C/Pthread semantics and the dynamic execution semantics of MCAPI functions (over 50 API calls). Thus, neither symbolic model checking methods nor model-based verification methods (*e.g.*, modeling C/Pthreads/MCAPI in say Promela) can help in verifying MCAPI applications. Dynamic verification methods were pioneered in Verisoft [4] precisely for this domain. In order to prevent the exponential growth in the number of potential thread interleavings (schedules), we will employ dynamic partial order reduction methods [5] that have been shown to be very effective in software verification.

Contributions: Our main contribution is the MCC model checker that verifies the connection-less message passing constructs of MCAPI using a reference implementation of the API. A large number of new concurrency APIs are being introduced to program future multi-core systems. We predict that each such API will require a DPOR-based [5] algorithm for verification. In our past work, we have built two such DPOR customizations for other APIs, namely Inspect [6] (for Pthreads) and ISP [7] (for MPI). This work builds on the strengths of ISP and Inspect but deviates from these tools in novel ways. For instance, in case of MPI, an explicit wildcard receive is provided whereas MCAPI, which borrows many ideas from MPI, does not do so. Therefore ISP's solution to accommodate the non-determinism by rewriting wildcard receive calls dynamically into specific receive calls so as to enforce a deterministic match with a sender at runtime, will not work in MCC's verification methodology. Unlike ISP, the scheduler of MCC also manages thread creation and thread join calls. MCC's verification methodology differs from Inspect with regard to the DPOR method that is employed. Inspect's DPOR mechanism does not support message passing. Other tools (*e.g.*, CHESS [8]) follow approaches to contain the number of interleavings by bounding the number of preemptions. In addition to being prone to bug omissions, preemption bounding is not a suitable approach for formal verification when message passing concurrency is involved because in message passing systems, many actions are largely independent of other actions (and hence commute) – and for these steps, exploring different interleavings is wasteful.

Funded collaoratively by NSF CCF 0903408/0903491 and SRC contract 2009-TJ-1993/1994

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

II. Verification of MCAPI

An MCAPI node serves as a logical abstraction for a thread of activity (which can be realized in multiple ways). It has multiple endpoints, each being a ⟨node id, port id⟩ pair. MCAPI also provides packet channels and scalar channels (not supported in MCC yet). Communication occurs within MCAPI through connection-less messages, connected packet channels, or connected scalar channels. All communications occur with respect to endpoints. Typical API calls include MCAPI_INITIALIZE, MCAPI_FINALIZE, as well as calls to create endpoints, send/receive messages in non-blocking mode, and later await for the completion of the send/receive. Details are available from [1].

A. Overview of MCC

We are building MCC even before public domain MCAPI applications are available. We also believe that MCC must be able to accept MCAPI library implementations produced by industries "as is," and use them to provide the execution semantics for MCAPI calls. We are currently employing a Pthread based reference implementation produced by the Multicore Associaton (MCA). All this ensures that (i) we will not waste time recreating the functionality of MCAPI (a very arduous task), and (ii) we can switch out one MCAPI library and switch in, say, a piece of silicon that purportedly realizes MCAPI (to see if we can find any new bugs by doing so during the *platform testing* mode).

In this paper, we focus exclusively on MCAPI's connection-less send and receive commands, and verify local assertions placed within threads, as well as deadlocks. We have also identified a list of default safety checks that are listed in [9] that we hope to incorporate in our future realizations of MCC.

MCAPI receive calls are non-deterministic in the presence of concurrent sends to a common endpoint. MCAPI receive calls only specify the destination endpoints on which the message should be received which precisely is the cause for non-determinism. Since receives are applied to endpoints and so are sends, it is possible that two sends could have a race in matching with a receive call. Our strategy to accommodate receive nondeterminism is: (i) have a dynamic algorithm to determine all senders that can match each receive, and (ii) then replay the execution of the entire MCAPI application, where for each replay we ensure that one of these sends matches the receive. The overall nature of execution control, along with DPOR is adapted from our group's tool ISP [10] and is illustrated in Figure 1.

The compile time instrumenter runs through the program body and converts all MCAPI calls and Pthread create and join calls to our own wrapper calls. The profiler intercepts these wrapper calls made by the user application, performs the required book keeping, and subsequently communicates with the verification scheduler. The verification scheduler can either give a *go-ahead* to a calling MCAPI thread or refrain from doing so to arrest the progress of that thread. The scheduler achieves two end goals. First, it manifests *independent* [11] thread steps according to a canonical order. *This ordering effects partial order reduction.* Second, for (non-deterministic) receives, the scheduler delays the processing of the receive till all sends that can potentially match the receive are dynamically discovered. It then replays the execution for these receives (these being the interesting *ample sets* [11]). The pseudocode of the scheduler is given in Figure 4.

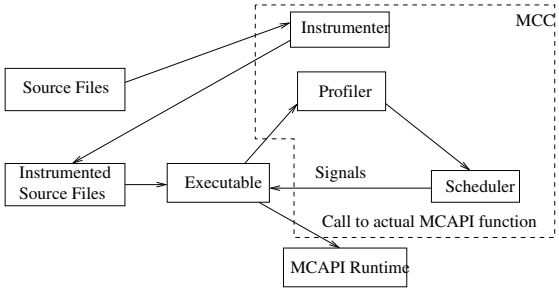


Fig. 1. MCC workflow

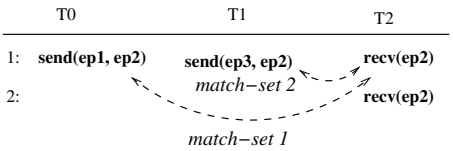


Fig. 2. MCAPI non-deterministic receive example

Figure 2 illustrates the motive behind our scheduler end-goal of delaying the processing of receive calls till all enabled matching sends are discovered. Suppose the scheduler discovers (as shown in Figure 2) that the send calls from threads T0 and T1 can both potentially match the receive posted by thread T2. Clearly, we must replay the execution for both these matches: in one execution, T0's call will match T2's first receive and T1's call will match T2's second receive (else there is a deadlock); and in the other execution, T1's call will match T2's first receive.

B. Illustration of MCC on an Example

Figure 3 illustrates a snippet of an executable MCAPI code prior to the instrumentation done by the MCC. The main thread in the example code spawns three threads. Threads with IDs 0 and 1 send a message to the thread with ID 2. The senders and the receiver have to explicitly create sending and receiving endpoints by issuing MCAPI create endpoint calls (lines 6,10). In order to get the address of the remote receiving endpoint, a *mcapi_get_endpoint* call is issued (line 11). Note that the *mcapi_get_endpoint* call is a blocking call. If the requested endpoint is never created then the *mcapi_get_endpoint* call may cause the system to deadlock. The MCC scheduler stores a list of endpoints that have already been created. An *mcapi_get_endpoint* call is instantly issued to the runtime if the associated endpoint has already been created, otherwise the scheduler delays the issuing of the call until the requested endpoint is created. The instrumentation component of MCC instruments the MCAPI communication calls with the same call names, however, the call names are now prefixed with "p". Additionally the POSIX thread create and join calls are also replaced by our own wrapper function calls. The

```
1:#define NUM_THREADS 3
2:#define PORT_NUM 1

3:void* run_thread (void *t) {
     ...
4:  mcapi_initialize(tid,&version,&status);
5:  if (tid == 2) {
6:  recv_endpt =
     mcapi_create_endpoint (PORT_NUM,&status);
7:  mcapi_msg_recv(recv_endpt,msg,
                  BUFF_SIZE,&recv_size,
                  &status);
8:  mcapi_msg_recv(recv_endpt,msg
                  BUFF_SIZE, &recv_size,
                  &status);
9:   } else {
10:  send_endpt = mcapi_create_endpoint
                  (PORT_NUM,&status);
11:  recv_endpt = mcapi_get_endpoint
                  (2,PORT_NUM,&status);
12:  mcapi_msg_send(send_endpt,recv_endpt,
                  msg,strlen(msg),
                  1,&status);
13:  }
14:  mcapi_finalize(&status);
15:}

16:int main () {
     ...
17:  for(t=0; t<NUM_THREADS; t++){
18:    rc = pthread_create(&threads[t],
       NULL, run_thread,
       (void *)&thread_data_array[t]);
19:  }
20:  for (t = 0; t < NUM_THREADS; t++) {
21:    pthread_join(threads[t],NULL);
22:  }
    ...
24:}
```

Fig. 3. MCAPI example C program

thread function bodies are instrumented with *thread_start* and *thread_end* calls which act as barrier points. The notion of introducing aforesaid calls is explained in Section II-C.

The wrapper calls are defined in MCC's profiler library. Subsequently, the executable is run under the controlled environment of the scheduler.

C. MCC Algorithm

Figure 4 in Section II-C explains the working of the scheduler. It assumes that all threads are created at the outset of the program, and thus is able to determine the total number of threads alive in the system (lines 2-15). Since the scheduling decisions are made once *all* the threads in the system have hit their local fence operations, it therefore becomes imperative to discover the total count of runnable threads in the system. The scheduler waits till all threads in the system have posted their respective blocking calls and have come to a halt (lines 18-28). Note that if a thread issues the *mcapi_finalize* or *thread_end* type calls then the count of alive threads is decremented (lines 25-27). At line 16, either the user spawned threads are blocked at their *thread_start* calls or they have yet to issue any MCAPI calls. Note that *thread_start* calls in the instrumented code act as barrier points that make sure that all threads are ready to run at the same state. The scheduler signals all the blocked threads to continue with their execution and continues to receive transitions from runnable threads until no thread is in a running state (lines 19-28). The scheduler then identifies *match-sets* (line 30) which consist of matching transitions that complete each other (*e.g.*, sends to a specific endpoint and receives from the same endpoint). The scheduler then liberates the threads forming the match-set (line 31). To identify the match-sets we identify the ample set [11] of transitions. The transitions in the ample set are then grouped as ⟨send, receive⟩ pairs based on compatible arguments. A deadlock is flagged if no match-sets are found and there are still runnable threads in the system (i.e., the *count* variable is still not 0).

```
 1:  GenerateInterleaving( ) {
 2:      while (1) { // Computes the total number of threads alive
 3:          t_i = receive_transition ();
 4:          if (t_i is thread_create) {
 5:              num_threads++;
 6:              signal go-ahead to thread_of(t_i);
 7:          }
 8:          if (t_i is thread_join || t_i is MCAPI communication call
                by thread "main") {
 9:              signal go-ahead to thread i;
10:              break;
11:          }
12:          if (t_i is thread_start) {
13:              update the status of thread i to blocked;
14:          }
15:      }// while (1) ends here

16:      count = num_threads;
17:      signal go-ahead to all the blocked threads;

18:      while (count) { // till no more threads are alive
19:          for each (runnable thread i)  {
20:              t_i = receive_transition from thread i;
21:              update transition_list of thread_of (t_i) in the current
                     state;
22:              if (t_i is of blocking_type) {
23:                  update the status of thread i to blocked;
24:              }
25:              if (t_i is of type finalize or thread_end) {
26:                  count --;
27:              }
28:          }
             // All threads are blocked here
29:          while (no thread is runnable) {
30:              find_matchset ();
31:              unblock the threads owning transitions in the above
                     match-set;
32:          }
33:      }// while (count) ends here
34:  }

35:  find_matchset( ) {
36:      if (ample_list of transitions is not empty); {
37:          for each (t_i in head element of the ample_list) {
38:              give a go-ahead to thread_of (i);
39:          }
40:          return;
41:      }
42:      flag that a deadlock found;
43:  }
```

Fig. 4. MCC scheduler algorithm

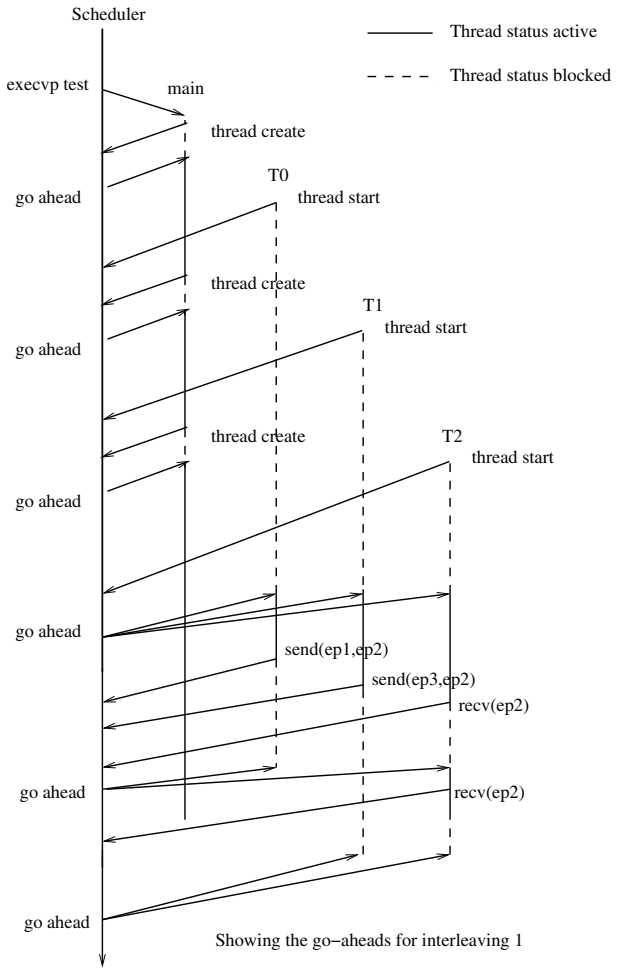


Fig. 5. Working of MCC on an example.

The procedure *GenerateInterleaving* is called in a loop until no more interleavings (replays) are left to explore. Different interleavings are verified by restarting the test target. The scheduler maintains a state consisting of a list of enabled transitions from each process. The ample set for each such state is computed in the first run. In subsequent runs, the per-state ample set is only updated by removing the match-sets that are signaled to proceed in that particular state of the run. In Figure 2, the match-set computed after the first run is the following:

- The match-set at state 1: $\{\langle send(ep1, ep2),\ recv(ep2)_1\rangle, \langle send(ep3, ep2),\ recv(ep2)_1\rangle\}$. Note that $recv(ep2)_1$ denotes the first receive call by T2.
- Assume that the scheduler signals a go-ahead to the entry $\langle send(ep1, ep2),\ recv(ep2)_1\rangle$. Thus, the match-set in state 1 is updated to $\{\langle send(ep3, ep2),\ recv(ep2)_1\rangle\}$.
- At state 2 the match-set formed is a singleton set $\{\langle send(ep3, ep2),\ recv(ep2)_2\rangle\}$. Note that $recv(ep2)_2$ denotes the second receive call by T2. After the scheduler signals a go-ahead, the match-set is reduced to an empty set.
- In the next run, the ample set at state 1 is again visited and the scheduler now decides to signal a go-ahead to $\langle send(ep3, ep2),\ recv(ep2)_1\rangle$.

Thus, two interleavings are sufficient for exhaustive exploration. The basic semantics guaranteed is that of non-overtaking (point to point FIFO ordering) as explained in [10]. Figure 5 shows the time-line diagram of an execution interleaving explored by running the instrumented example code from Figure 3. The scheduler starts running by executing the test target. The main thread of the test target issues the thread create calls and subsequently gets blocked until it receives a go-ahead signal from the scheduler. Note that all threads created by the main thread are blocked on their respective *thread_start* calls. The scheduler after assessing the total thread count, signals a go-ahead to all such threads blocked on the *thread_start* calls. The scheduler computes a match-set once all the threads have blocked on their respective MCAPI operations. It then selects one entry from the match-set and signals a go-ahead to the participating threads (in Figure 3, it is $\langle T0,\ T2\rangle$ followed by $\langle T1,\ T2\rangle$).

III. Results and Conclusions

We have developed the first dynamic verifier that handles a subset of MCAPI calls using only publicly available MCAPI resources. We have developed a scheduler with a robust runtime control method building on past work. MCC has successfully handled several simple example programs. Deterministic programs are verified in one interleaving for the absence of deadlocks and safety violation assertions. The example program from Figure 3 was verified in 2 interleavings with no deadlocks found. Through active collaborations with the MCA, we are developing public-domain MCAPI benchmark applications. We are also extending MCC to cover the full gamut of MCAPI calls, with an approach (under testing) for handling connection-less oriented non-blocking calls. Future research also includes programs that use shared memory and message passing for this mixed domain, we are expecting safe concurrency patterns to emerge, which MCC will then exploit.

We acknowledge Jay Bhadra of Freescale and Neha Rungta and Topher Fischer of BYU for their help on this work.

References

[1] Multicore Association. http://www.multicore-association.org.
[2] MPI Forum. http://www.mpi-forum.org
[3] http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html
[4] Patrice Godefroid. Software Model Checking: TheVeriSoft Approach. FMSD 2005, vol 26-2, 77-101.
[5] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *POPL*, 110–121, 2005.
[6] Yu Yang, Xiofang Chen, Ganesh Gopalakrishnan, R.M. Kirby. Distributed Dynamic Pratial Order Reduction Based Verification. SPIN 2007, 58-75.
[7] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. PPoPP 2008. 285-286.
[8] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI 2007, 446-455.
[9] http://www.cs.utah.edu/formal_verification/mediawiki/index.php/MCAPI
[10] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. CAV 2008, 66-79.
[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

Generalized, Efficient Array Decision Procedures

Leonardo de Moura
Microsoft Research, USA, leonardo@microsoft.com

Nikolaj Bjørner
Microsoft Research, USA, nbjorner@microsoft.com

***Abstract*—The theory of arrays is ubiquitous in the context of software and hardware verification and symbolic analysis. The basic array theory was introduced by McCarthy and allows to symbolically representing array updates. In this paper we present *combinatory array logic*, CAL, using a small, but powerful core of combinators, and reduce it to the theory of uninterpreted functions. CAL allows expressing properties that go well beyond the basic array theory. We provide a new efficient decision procedure for the base theory as well as CAL. The efficient procedure serves a critical role in the performance of the state-of-the-art SMT solver Z3 on array formulas from applications.**

I. Introduction

As part of formulating a programme of a mathematical theory of computation McCarthy [1] proposed a basic theory of arrays. The basic theory characterizes functions *store* and the binary selector $\_[\_]$ using two axioms: $\forall a, i, v \;.\; \mathit{store}(a, i, v)[i] \simeq v$ and $\forall a, i, j, v \;.\; i \simeq j \vee \mathit{store}(a, i, v)[j] \simeq a[j]$.

In this paper we develop an efficient saturation procedure for the basic (extensional) array theory as well as a powerful extension that we call *combinatory array logic* (CAL). Besides the *store* combinator the extension uses two new main combinators K (inspired by combinatory logic) and map_f (that maps f on arrays). They have the characteristics:

$$\begin{aligned} K(v)[i] &= v \\ \mathit{map}_f(a_1, \ldots, a_n)[i] &= f(a_1[i], \ldots, a_n[i]) \end{aligned}$$

Ground satisfiability in the resulting theory is shown to be NP-complete. Our procedures are presented as inference rules. A useful contribution of this paper is strong filters for restricting the application of these rules while retaining completeness. The results are developed in the context of strongly disjoint theories, where finite domains are easy to handle. We show how default values of arrays can be reflected back into the array theory, but this construction is very sensitive to domain sizes. A technical report, available from the authors, describes CAL with the identity operator I.

The ideas described in this paper were already implemented in the version of the SMT solver Z3 submitted to the SMT 2008 (http://www.smtcomp.org). Z3 won the QF_AUFLIA division (arrays, uninterpreted functions and linear integer arithmetic), and was 25 times faster than the second place (Barcelogic). In the QF_A division, Z3 finished in second place, but this division consisted only of trivial artificial problems. The winner (Barcelogic) total runtime was 13.5 secs and Z3 was 17.3 secs. In Section VI, we also compare the performance of Z3 with and without using some of the proposed filters.

A. Related Work

Decision procedures for non-extensional theories of arrays with Presburger arithmetic constraints appeared in the early 80's [2], [3]. The theory remains important in the context of formal verification of hardware [4], [5].

A decision procedure for the theory of extensional arrays is given in [6]. It uses constrained equations between arrays to capture when arrays are equal except possibly on a finite set of indices. Rewriting approaches in the context of super-position are presented in [7] and [8]. An approach that also uses constrained equations is developed in [9]. It produces clauses with constrained equations, but a distinguishing feature of this system is that it uses the current congruence closure model to guide the search, thereby avoiding potentially redundant cases.

The theory of equality and uninterpreted functions is in a sense a base theory for the theory of arrays. Array access $a[i]$ can be treated as a binary uninterpreted function, and array updates can be compiled away using a finite set of instances. This was recognized for the theory of arrays as well as a number of other theories in [10]. The reduction approach is the basis of several implementations of the theory of arrays, including Yices [11], Z3 [12], and analyzed in DPT [13]. STP [14] also uses a reduction approach, and furthermore observes that it can be important to delay rewriting array read/write terms into conditional statements. As an alternative to eliminating array writes, [15] considers eliminating reads in favor of writes. The resulting procedure handles especially well cases where arrays obtained by multiple non-interfering overwrites are compared. The *map* and *array property fragments* [16] are classes of first-order formulas that can express array properties involving some arithmetic. Extensions are studied in [17], including a unary map operator. An entirely different approach to arrays represents models of arrays as regular automata [18]. They can decide formulas that use offset arithmetic on array indices.

In comparison our paper offers a general setting for optimized array decision procedures based on inference rules.

II. Preliminaries

We consider a many-sorted language. A *signature* Σ is a triple $(\Sigma^S, \Sigma^F, \Sigma^P)$ where Σ^S is a set of sorts, Σ^F is a set of function symbols, and Σ^P is a set of predicate symbols (each endowed with the corresponding arity and sort). We assume that, for each sort σ, the equality $\simeq_\sigma$ is a symbol that does not occur in Σ and that is always interpreted as the identity relation over (the interpretation of) σ. As a notational convention, we will always omit the subscript. We call 0-arity function symbols *constant* symbols, and 0-arity predicate

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

symbols *propositions*. Σ-*atoms*, Σ-*literals*, Σ-*clauses*, and Σ-*formulas* are defined in the usual way. A set of Σ-literals is called a Σ-constraint. Terms, literals, clauses and formulas are called *ground* when no variable appears in them. A *sentence* is a formulas in which free variables do not occur. A *CNF formula* is a conjunction $C_1 \wedge \ldots \wedge C_n$ of clauses. We will write CNF formulas as set of clauses. We use a, b, i, j, v and w for constants, where a and b are used for array constants, i and j for array indices, and v and w for array values. We use f for function symbols, p and q for predicate symbols, σ and τ for sorts, C for clauses, and φ for formulas. We use $v\colon\sigma$ to denote that constant symbol v has sort σ, and $f\colon(\sigma_1,\ldots,\sigma_n)\to\tau$ to denote that function symbol f has arity n, argument sorts $\sigma_1,\ldots,\sigma_n$, and result sort τ. Given two signatures Σ_1 and Σ_2, $\Sigma_1\cup\Sigma_2$ and $\Sigma_1\subseteq\Sigma_2$ are defined as usual, we say Σ_1 and Σ_2 are *disjoint* if $\Sigma_1^F\cap\Sigma_2^F=\emptyset$ and $\Sigma_1^P\cap\Sigma_2^P=\emptyset$, and *strongly disjoint* if they are disjoint and $\Sigma_1^S\cap\Sigma_2^S=\emptyset$.

We use the standard notion of a Σ-structure M. It consists of a non-empty pairwise disjoint domains $|M|_\sigma$ for every sort σ, and a sort and arity-matching interpretation of the function and predicate symbols in Σ. We use ι and ν for elements of a domain $|M|_\sigma$. We use $M(f)$ (resp. $M(p)$) to denote the interpretation of the function (resp. predicate) symbol f (resp. p). The interpretation of an arbitrary term t is denoted by $M[t]$, and is defined in the standard way. The truth of a Σ-formulas φ, denoted by $M[\varphi]$, is also defined in the standard way. If $\Sigma_0\subseteq\Sigma$ and M is a Σ-structure, the Σ_0-*reduct* of M is the Σ_0-structure $M{\downarrow_{\Sigma_0}}$ obtained from M by forgetting the interpretation of the symbols from $\Sigma\setminus\Sigma_0$.

A collection of Σ-sentences is a Σ-theory. The free theory $T_\emptyset$ over a signature Σ is the first-order theory with an empty set of Σ-sentences. Let T_1 be a Σ_1-theory and T_2 be a Σ_2-theory. Then, T_1 and T_2 are *disjoint* (resp. *strongly disjoint*) if Σ_1 and Σ_2 are disjoint (resp. strongly disjoint). The combined theory $T_1\oplus T_2$ is a $(\Sigma_1\cup\Sigma_2)$-theory composed by the union of the Σ_1 and Σ_2-sentences. The *constraint satisfiability problem* for a theory T, also called the T-satisfiability problem, is the problem of deciding whether a Σ-constraint is satisfiable in a model of Σ-theory T. The constraint may contain variables, since these variables may be replaced by fresh constants, we can define the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals, in an expanded signature $\Sigma_\star$, is true in a $\Sigma_\star$-structure whose Σ-reduct is a model of T. The *satisfiability problem* can be similarly defined for ground CNF formulas.

III. A Simple Core Solver

The array theory decision procedures, proposed in this paper, are defined on top of a core solver as a set of inference rules. The basic architecture of the core solver is the usual one used in state-of-the-art SMT solvers, where a SAT solver is integrated with a decision procedure for the constraint satisfiability problem for a Σ-theory T [19]. In our core solver, the *core theory* T_{core} is the combined theory $T_\emptyset\oplus T_1\oplus\ldots\oplus T_k$, where for each $i,j\in\{1,\ldots,k\}$, T_i and T_j are strongly disjoint, and $T_\emptyset$ and T_i are disjoint. This restriction admits a very simple combination method where the theories T_i's can be non-stably infinite and non-convex. Our combination method uses the *model-based theory combination* [20]. In the rest of the paper, we assume that one of the theories T_i's is the Σ_b-theory T_b of Boolean terms, where $\Sigma_b^S=\{\mathsf{bool}\}$, $\Sigma_b^F=\{\top\colon\mathsf{bool},\bot\colon\mathsf{bool}\}$, $\Sigma_b^P=\emptyset$, and it contains the following two axioms:

$$\top\not\simeq\bot,\qquad \forall x\colon\mathsf{bool}\,.\,x\simeq\top\vee x\simeq\bot$$

In our actual solver, the other theories in T_{core} are: *arithmetic*, *bit-vectors* and *scalar values*. For each $i\in\{1,\ldots,k\}$, let Σ_i be the signature of theory T_i, and let $\Sigma_\emptyset$ be the signature of $T_\emptyset$. Recall that the signature of $T_\emptyset$ is not fixed a priori, and w.l.o.g. we assume $\Sigma_1^S\cup\ldots\cup\Sigma_k^S\subseteq\Sigma_\emptyset^S$. We say a function (resp. predicate) symbol f (resp. q) is *interpreted* if $f\in\Sigma_1^F\cup\ldots\cup\Sigma_k^F$ (resp. $q\in\Sigma_1^P\cup\ldots\cup\Sigma_k^P$). Otherwise, we say the symbol is *uninterpreted*. We also assume uninterpreted predicates $q(v_1,\ldots,v_n)$ are represented as $f_q(v_1,\ldots,v_n)\simeq\top$. In the core solver, CNF formulas are represented in flattened form. A *CNF flattened formula* comprises of a sequence of definitions of the form:

$$v\equiv f(v_1,\ldots,v_n),\qquad p\equiv q(v_1,\ldots,v_n),\qquad p\equiv v\simeq w$$

and clauses of the form $l_1\vee\ldots\vee l_n$, where f is a function symbol, p is an uninterpreted proposition, q is a predicate symbol, $v,w,v_1,\ldots,v_n$ are uninterpreted constants, each v is never defined *after* it is used, and each l_i is of the form p or $\neg p$. The constant v and proposition p, above, should be viewed as *names* for terms and atoms respectively. In an actual implementation, they are essentially pointers to these terms and atoms.

Example 1: The CNF formula $v\simeq w\wedge(v\geq w\vee f(v-w)\simeq 0)$ can be represented in flattened form as:

$$\begin{array}{l} p_1\equiv v\simeq w,\quad p_2\equiv v\geq w,\quad v_1\equiv v-w,\\ v_2\equiv f(v_1),\quad v_3\equiv 0,\quad p_3\equiv v_2\simeq v_3,\end{array}\quad ;\ p_1,\quad p_2\vee p_3$$

The SAT solver, in our core solver, uses a DPLL based algorithm to build an assignment for all propositions. For each $i\in\{1,\ldots,k\}$, let $\mathfrak{S}_i$ be a decision procedure for theory T_i, and let $\mathfrak{S}_\emptyset$ be a decision procedure for the free theory $T_\emptyset$. In our implementation, $\mathfrak{S}_\emptyset$ is based on the congruence closure algorithm described in [21]. The state Γ of the core solver is composed by a propositional assignment, a set of definitions and clauses $F(\Gamma)$, and the states of procedures $\mathfrak{S}_i$'s and $\mathfrak{S}_\emptyset$. We use $\Gamma(p)$ to denote the assignment of proposition p in state Γ. When the SAT solver assigns a proposition $p\equiv q(v_1,\ldots,v_n)$ and $q\in\Sigma_i^P$, then procedure $\mathfrak{S}_i$ is notified. If $p\equiv v\simeq w$, then $\mathfrak{S}_\emptyset$ is notified, and if v has sort $\sigma\in\Sigma_i^S$, then procedure $\mathfrak{S}_i$ is also notified. The procedure $\mathfrak{S}_\emptyset$ maintains an equivalence relation $\sim_\Gamma$, which is the smallest equivalence relation that contains $\{(v,w)\mid p\equiv v\simeq w\in\Gamma,\text{ and }\Gamma(p)=\mathsf{true}\}$. As usual, the relation $\sim_\Gamma$ can be implemented using a union-find data-structure. The procedure $\mathfrak{S}_\emptyset$ also maintains the relation $\not\sim_\Gamma$ defined as $\{(v,w)\mid p\equiv v'\simeq w'\in\Gamma,\ \Gamma(p)=\mathsf{false},\ v\sim_\Gamma v',\ w\sim_\Gamma w'\}$. As a notational convention we will always omit the subscript in

$\sim_\Gamma$ and $\not\sim_\Gamma$. Inference rules are written as:

$$\frac{\alpha_1, \ldots, \alpha_n}{C_1, \ldots, C_m}$$

where $\alpha_1, \ldots, \alpha_n$ are the *antecedents*, and should be viewed as queries to the current state Γ of the core solver. We use antecedents of the form: $a \equiv f(v_1, \ldots, v_n)$ (meaning: the definition is in Γ), $v \sim w$ (meaning: v and w are equivalent in Γ), $v{:}\,\sigma$ (meaning: Γ contains a constant v of sort σ), and $\Gamma(p) = \mathsf{false}$. The *consequents* $C_1, \ldots, C_n$ are clauses, not necessarily in flattened form, that should be added to the next state. For example, if the consequent is of the form $a[i] \simeq v$, it should be interpreted as $v' \equiv a[i], \quad p \equiv v' \simeq v; \quad p$. Note that new definitions are not created if they already exist in current state Γ. We use $\Gamma_1 \vdash_\gamma \Gamma_2$ to denote that inference rule γ was applied to state Γ_1 producing a new state Γ_2. We say an inference rule γ is *sound* with respect to a theory T if for all states Γ_1 and Γ_2 such that $\Gamma_1 \vdash_\gamma \Gamma_2$, we have $F(\Gamma_1)$ is equisatisfiable to $F(\Gamma_2)$ modulo theory T. A inference rule γ is *saturated* at state Γ if Γ already contains any consequent of γ, or if the consequents are already satisfied by the (partial) propositional assignment in Γ. In our implementation, the procedure $\mathfrak{S}_\emptyset$ uses the congruence inference rule:

$$\frac{w_1 \equiv f(v_1, \ldots, v_n),\ w_2 \equiv f(v'_1, \ldots, v'_n),\ v_1 \sim v'_1, \ldots, v_n \sim v'_n}{w_1 \simeq w_2}$$

Each procedure $\mathfrak{S}_i$ builds an interpretation (*candidate model*) for each constant $v{:}\,\sigma$ s.t. $\sigma \in \Sigma_i^S$. The combination method requires that the procedures agree on equalities between (uninterpreted) constants. For this purpose, the model-based theory combination, uses the models M_i that are built by each procedure $\mathfrak{S}_i$. Given two constants v and w, such that $M_i(v) = M_i(w)$, the procedure creates a definition $p_{v,w} \equiv v \simeq w$ (if it does not exist already), and $p_{v,w}$ is assigned to true in the SAT solver. This assignment is essentially a *guess* (i.e., decision), if this assignment triggers an inconsistency, then backtracking is used to fix the model. Many optimizations are possible [20] in the architecture described above, but they are beyond the scope of this paper. We say Γ is a *satisfiable final state* if all inference rules are saturated, all propositions are assigned, all clauses are satisfied, all constants $v{:}\,\sigma$ have an interpretation when $\sigma \in \Sigma_i^S$ for some $i \in \{1, \ldots, k\}$, and none of the procedures $\mathfrak{S}_i$'s and $\mathfrak{S}_\emptyset$ detected an inconsistency.

Example 2: Consider the following CNF formula, where $f{:}\,\mathsf{bool} \to \sigma$, $v{:}\,\mathsf{bool}$ and $w{:}\,\sigma$.

$$f(\top) \simeq w \ \wedge \ f(\bot) \simeq w \ \wedge \ f(v) \not\simeq w$$

This formula is unsatisfiable, and the core solver will detect it. The procedure $\mathfrak{S}_b$ will try to assign an interpretation for v because it has sort bool, but an inconsistency is detected (using the congruence rule) when it tries to assign v to $\top$ or $\bot$. Note that none of the procedures $\mathfrak{S}_i$ had to exchange cardinality constraints.

IV. Array Theory

The array theory T_A with signature Σ_A is parametric in the context of many-sorted logic. That is, given a non-empty set of sorts $\mathcal{S}$, Σ_A^S is the least set such that:

$$\mathsf{idx}\ \frac{a \equiv \mathit{store}(b, i, v)}{a[i] \simeq v}$$

$$\Downarrow\ \frac{a \equiv \mathit{store}(b, i, v), \quad w \equiv a'[j], \quad a \sim a'}{i \simeq j \vee a[j] \simeq b[j]}$$

$$\Uparrow\ \frac{a \equiv \mathit{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b'}{i \simeq j \vee a[j] \simeq b[j]}$$

$$\mathsf{ext}\ \frac{a{:}\,(\sigma \Rightarrow \tau), \quad b{:}\,(\sigma \Rightarrow \tau)}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

Fig. 1. Array theory basic inference rules.

1) $\mathcal{S} \subset \Sigma_A^S$
2) $\sigma \in \Sigma_A^S$ and $\tau \in \Sigma_A^S$ implies $(\sigma \Rightarrow \tau) \in \Sigma_A^S$.

We say sorts of the form $(\sigma \Rightarrow \tau)$ are *array sorts* with *index sort* σ, and *value sort* τ. For each array sort $(\sigma \Rightarrow \tau)$, Σ_A^F contains the function symbols $\_[\_]{:}\,((\sigma \Rightarrow \tau), \sigma) \to \tau$, and $\mathit{store}{:}\,((\sigma \Rightarrow \tau), \sigma, \tau) \to (\sigma \Rightarrow \tau)$. There are no predicates, so $\Sigma_A^P = \emptyset$. We say $\_[\_]$ is the *array read* operation, and *store* the *array update* operation. The following scheme axiomatizes these two operators:

$$\forall a{:}\,(\sigma \Rightarrow \tau),\ i{:}\,\sigma,\ v{:}\,\tau\,.\,\mathit{store}(a, i, v)[i] \simeq v$$
$$\forall a{:}\,(\sigma \Rightarrow \tau),\ i{:}\,\sigma,\ j{:}\,\sigma,\ v{:}\,\tau\,.\,i \simeq j \vee \mathit{store}(a, i, v)[j] \simeq a[j]$$

We say that the function symbol *store* is an array combinator, that is, operations that build new arrays. Later, we define a richer set of array combinators. The following scheme is called the extensionality axiom scheme. Informally, it states that if two arrays store the same value at index i, for each index i, then they are equal.

$$\forall a{:}\,(\sigma \Rightarrow \tau),\ b{:}\,(\sigma \Rightarrow \tau)\,.\,\exists i{:}\,\sigma\,.\,a[i] \not\simeq b[i] \vee a \simeq b$$

Fig. 1 contains a basic set of inference rules for the array theory. Let us explain the first rules informally. Rule idx adds the assertion $a[i] \simeq v$ for every occurrence of a definition $a \equiv \mathit{store}(b, i, v)$. Rule $\Downarrow$ propagates read over a *store*. It fires if a is defined as a *store* and in the current state a is equivalent to a', where a' occurs in a read. It adds a clause forcing either the index j to be equal to the update index i, or the contents of a to agree with b on j. The clause is a tautology in the theory of arrays, it does not depend on $a \sim a'$. The other rules should be interpreted in a similar way. Later, we propose many refinements.

Theorem 3 (Soundness): idx, $\Downarrow$, $\Uparrow$, ext are sound.

Proof: The rules idx, $\Downarrow$, $\Uparrow$ are just instantiating the array axioms. The rule ext is instantiating the extensionality axiom by using a fresh skolem constant $k_{a,b}$. ■

Theorem 4 (Termination): idx, $\Downarrow$, $\Uparrow$, ext are terminating.

Proof: None of the rules create definitions of the form $a \equiv \mathit{store}(b, i, v)$. Thus, rule idx can only be applied once for each occurrence of *store* in the input. Assume the input formula has n array constants ($a{:}\,(\sigma \Rightarrow \tau)$), and m definitions of the form $v \equiv a[j]$. Then, rule ext can be applied at most n^2 times, and at most n^2 skolem constant $k_{a,b}$ are created. The rules $\Downarrow$ and $\Uparrow$ can be applied at most $(n^2 + m)n$ times. ■

Definition 5 (Map): Given sets S_σ and S_τ, a *map* from S_σ to S_τ is a finite set of pairs (ι, ν) where $\iota \in S_\sigma$ and $\nu \in S_\tau$. We say the map G is *functional* iff for all (ι_1, ν_1) and (ι_2, ν_2) in G, $\iota_1 = \iota_2$ implies that $\nu_1 = \nu_2$.

Theorem 6 (Completeness): idx, $\Downarrow$, $\Uparrow$, ext are complete.

Proof: Assume all rules are saturated in the satisfiable final state Γ, and let M be the model produced by the core solver for this final state. Note that symbols *store* and $\_[\_]$ are considered to be uninterpreted in the core solver. The core solver guarantees that for any pair of constants v_1 and v_2, $v_1 \sim v_2$ iff $M(v_1) = M(v_2)$. Our goal is to build a model M^λ that satisfies Γ and the array axioms. For every non array sort sort σ, $|M^\lambda|_\sigma = |M|_\sigma$. The domain for the array sorts is defined inductively. Let σ' be an array sort of the form $(\sigma \Rightarrow \tau)$, then $|M^\lambda|_{\sigma'}$ is the set of functions from $|M^\lambda|_\sigma$ to $|M^\lambda|_\tau$. The interpretation for each $\_[\_]\colon((\sigma \Rightarrow \tau), \sigma) \to \tau$ is just the function application. More formally, given $\rho \in |M^\lambda|_{(\sigma \Rightarrow \tau)}$ and $\iota \in |M^\lambda|_\sigma$, $M^\lambda(\_[\_])(\rho, \iota) = \rho(\iota)$. The interpretation for each *store*$\colon((\sigma \Rightarrow \tau), \sigma, \tau) \to (\sigma \Rightarrow \tau)$ is $M^\lambda(\mathit{store})(\rho, \iota, \nu) = \rho'$, where ρ' is the function:

$$\rho'(x) = \begin{cases} \nu & \text{if } x = \iota, \\ \rho(x) & \text{otherwise.} \end{cases}$$

It is easy to check that the interpretations for $\_[\_]$ and *store* satisfy the array axioms. Our next goal is to assign an interpretation to all constants in Γ such that:

1) For any pair of constants v_1 and v_2 in Γ, $M(v_1) = M(v_2)$ iff $M^\lambda(v_1) = M^\lambda(v_2)$. We say this is the *equivalence property*.
2) The interpretation of constants satisfies all definitions of the form $a \equiv \mathit{store}(b, i, v)$ and $v \equiv a[i]$ in Γ.

Let $\sqsubset$ be a total order on sorts such that for all array sorts $(\sigma \Rightarrow \tau)$, $\sigma \sqsubset (\sigma \Rightarrow \tau)$ and $\tau \sqsubset (\sigma \Rightarrow \tau)$. We define the interpretation for the constants using $\sqsubset$. That is, if $\sigma_1 \sqsubset \sigma_2$, then we define the interpretation for all constants $a_1\colon\sigma_1$ before defining the interpretation for any constant $a_2\colon\sigma_2$. Moreover, when we construct the interpretation for $a_2\colon\sigma_2$ we assume that the equivalence property holds for all constants $a_1\colon\sigma_1$ where $\sigma_1 \sqsubset \sigma_2$. The "base case" is easy, for each constant $v\colon\sigma$ in Γ, such that σ is not an array sort, $M^\lambda(v) = M(v)$. We also define $M^\lambda(f) = M(f)$ for every interpreted function symbol f. For each sort σ, Let δ_σ be an arbitrary element of $|M^\lambda|_\sigma$. Now, we define an interpretation for an array constant $a\colon(\sigma \Rightarrow \tau)$ assuming that the interpretation for all constants $i\colon\sigma$ and $v\colon\tau$ was already defined. First, we define a map $\mathsf{graph}(a)$ as the set $\{(M^\lambda(i), M^\lambda(v)) \mid v \equiv a'[i] \in \Gamma,\ a \sim a'\}$. The map $\mathsf{graph}(a)$ is functional, because the equivalence property holds for all constants $i\colon\sigma$ and $v\colon\tau$; and for any two entries $v_1 \equiv a_1[i_1]$ and $v_2 \equiv a_2[i_2]$ the core solver guarantees that $M(a_1) = M(a_2)$ and $M(i_1) = M(i_2)$ implies that $M(v_1) = M(v_2)$. Then, we define $M^\lambda(a)$ as:

$$M^\lambda(a)(\iota) = \begin{cases} \nu & \text{if } (\iota, \nu) \in \mathsf{graph}(a), \\ \delta_\sigma & \text{otherwise.} \end{cases}$$

Now, we show that for any array constants $a\colon(\sigma \Rightarrow \tau)$ and $b\colon(\sigma \Rightarrow \tau)$, the equivalence property holds:

1) If $a \sim b$, then by construction $M^\lambda(a) = M^\lambda(b)$.
2) If $a \not\sim b$, then rule ext guarantees that $\mathsf{graph}(a)$ contains $(M^\lambda(k_{a,b}), \nu_1)$ and $\mathsf{graph}(b)$ contains $(M^\lambda(k_{a,b}), \nu_2)$ such that $M^\lambda(\nu_1) \neq M^\lambda(\nu_2)$. Therefore, $M^\lambda(a) \neq M^\lambda(b)$.

It is easy to check that the definitions of the form $v \equiv a[i]$ are satisfied by M^λ. Now, we show that M^λ also satisfies all definitions of the form $a \equiv \mathit{store}(b, i, v)$. Recall that all rules are saturated in the final state. First, rule idx guarantees that $M^\lambda(a)(M^\lambda(i)) = M^\lambda(v)$. Now, let $\mathsf{index}(a)$ be the set $\{\iota \mid (\iota, \nu) \in \mathsf{graph}(a)\}$. The rules $\Uparrow$ and $\Downarrow$ guarantee that $\mathsf{index}(a) = \mathsf{index}(b) \cup \{M^\lambda(i)\}$, and $M^\lambda(a)(\iota) = M^\lambda(b)(\iota)$ for every $\iota \in \mathsf{index}(a) \setminus \{M^\lambda(i)\}$. Finally, we have $M^\lambda(a)(\iota) = M^\lambda(b)(\iota) = \delta_\sigma$ for every $\iota \notin \mathsf{index}(a)$. Therefore, every definition of the form $a \equiv \mathit{store}(b, i, v)$ is satisfied. The construction of the interpretation $M^\lambda(f)$, for each uninterpreted function symbol f in Γ, is similar to the one used for array constants. The only difference is that $\mathsf{graph}(f)$ is a tuple of size arity$(f) + 1$ instead of being a pair. It guarantees that all definitions of the form $v \equiv f(w_1, \ldots, w_n)$ are satisfied by M^λ. Note that the equivalence property guarantees that for every definition of the form $p \equiv v \simeq w$ in Γ, $M[v \simeq w]$ iff $M^\lambda[v \simeq w]$, and consequently for every clause C in Γ, $M[C]$ iff $M^\lambda[C]$. Thus, M^λ satisfies all array axioms, definitions and clauses in Γ. ■

A. Redundant Axioms

The rules $\Downarrow$, $\Uparrow$, ext produce clauses of the form:

$$i \simeq j \vee a[j] \simeq b[j] \tag{1}$$

$$a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}] \tag{2}$$

The proof of Theorem 6 makes it clear that it is unnecessary to add the clauses of the form (1) to Γ, when Γ already contains a clause $i' \simeq j' \vee a'[j'] \simeq b'[j']$ such that $a \sim a'$, $b \sim b'$, $i \sim i'$, and $j \sim j'$. Similarly, it is unnecessary to add (2) to Γ, when Γ already contains a clause $a' \simeq b' \vee a'[k_{a',b'}] \not\simeq b'[k_{a',b'}]$ such that $a \sim a'$ and $b \sim b'$. Thus, in our implementation, we use a data-structure for storing a set of tuples (a, b, i, j) for (1), and a set of tuples (a, b) for (2). Given a tuple t, this data-structure provides a constant time function for checking whether the data-structure contains a tuple congruent to t or not. Before including (1) and (2) into Γ, we check whether the data-structure already contains a congruent tuple. If it does, we discard the new clause. Otherwise, we include it into Γ and update the data-structure. This data-structure is similar to the hashtable used to implement congruence closure procedures [21].

B. Restricted Extensionality

In the proof of Theorem 6, rule ext is used to guarantee that for all array constants a_1 and a_2:

$$M(a_1) \neq M(a_2) \text{ implies } M^\lambda(a_1) \neq M^\lambda(a_2) \tag{3}$$

$$\mathsf{ext}_{\not\simeq}\ \frac{p \equiv a \simeq b, \quad \Gamma(p) = \mathsf{false}}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

$$\mathsf{ext}_{r}\ \frac{a{:}\,(\sigma \Rightarrow \tau), \quad b{:}\,(\sigma \Rightarrow \tau), \quad \{a, b\} \subseteq \mathsf{foreign}}{a \simeq b \vee a[k_{a,b}] \not\simeq b[k_{a,b}]}$$

Fig. 2. Restricted extensionality inference rules.

when proving the *equivalence property*: for any pair of constants v_1 and v_2 in Γ, $M(v_1) = M(v_2)$ iff $M^\lambda(v_1) = M^\lambda(v_2)$. We say an array constant a is *foreign* iff there is a b such that $a \sim b$, and b occurs as the argument of an uninterpreted function symbol f, or as the index of an array read $v \equiv a'[b]$. Observe that (3) is only needed for showing that:

1) $\mathsf{graph}(a')$ and $\mathsf{graph}(f)$ are functional when $\mathsf{index}(a)$ and $\mathsf{index}(f)$ contain $M^\lambda(a_1)$ and $M^\lambda(a_2)$. That is, a_1 and a_2 are foreign.
2) $M[a \simeq b] = \mathsf{false}$ implies $M^\lambda[a \simeq b] = \mathsf{false}$.

So, this observation suggests a simple optimization where ext is only applied to pairs of array constants a and b when: a and b are foreign, or $a \simeq b$ is assigned to false by the core solver.

Definition 7 (Foreign): Given a state Γ, the set $\mathsf{foreign}$ of *foreign constants* is the least set s.t.:

1) $v \equiv f(\ldots, a, \ldots)$ and $a{:}\,(\sigma \Rightarrow \tau)$ implies $a \in \mathsf{foreign}$,
2) $v \equiv a[b]$ and $b{:}\,(\sigma \Rightarrow \tau)$ implies $b \in \mathsf{foreign}$,
3) $a \sim b$ and $a \in \mathsf{foreign}$ implies $b \in \mathsf{foreign}$.

Fig. 2 contains the set of rules for implementing this refinement.

Theorem 8: The rules idx, $\Downarrow$, $\Uparrow$, $\mathsf{ext}_{\not\simeq}$ and ext_r are sound, terminating and complete.

Another optimization is based on the observation that it is unnecessary to add (2) to Γ, if a and b already store different values at some index. More formally, we have:

Definition 9 (Already Disequal): Given a state Γ, $(a, b) \in \mathsf{already\text{-}diseq}$ iff there are two definitions $v_1 \equiv a_1[i_1]$ and $v_2 \equiv a_2[i_2]$ in Γ such that $v_1 \not\sim v_2$, $a \sim a_1$, $b \sim b_1$, and $i_1 \sim i_2$.

C. Restricted $\Uparrow_r$

Definition 10 (Linearity): Given a state Γ, the set $\mathsf{non\text{-}linear}$ of *non-linear constants* is the least set such that:

1) $a_1 \equiv \mathit{store}(b_1, i_1, v_1)$, $a_2 \equiv \mathit{store}(b_2, i_2, v_2)$, a_1 is not a_2 and $a_1 \sim a_2$ implies $\{a_1, a_2\} \subseteq \mathsf{non\text{-}linear}$,
2) $a \equiv \mathit{store}(b, i, v)$ and $a \in \mathsf{non\text{-}linear}$ implies $b \in \mathsf{non\text{-}linear}$,
3) $a \in \mathsf{non\text{-}linear}$ and $a \sim b$ implies $b \in \mathsf{non\text{-}linear}$.

We say a is *linear* if $a \notin \mathsf{non\text{-}linear}$.

In many software verification applications, we observed that the set $\mathsf{non\text{-}linear}$ is very small. This observation suggests a simple optimization, where rule $\Uparrow$ is only applied to array constants in the set $\mathsf{non\text{-}linear}$. Given a map m and a constant j, we use $m \setminus \{j\}$ to denote the set $\{(\iota, \nu) \mid (\iota, \nu) \in m,\ \iota \neq M^\lambda(j)\}$. The basic idea is to use $\mathsf{graph}(b) \setminus \{i\}$ to complete the map $\mathsf{graph}(a)$ whenever Γ contains a definition of the form $a \equiv \mathit{store}(b, i, v)$ and b is linear. Fig. 3 contains the restricted version of $\Uparrow$.

$$\Uparrow_r\ \frac{a \equiv \mathit{store}(b, i, v), \quad w \equiv b'[j], \quad b \sim b', \quad b \in \mathsf{non\text{-}linear}}{i \simeq j \vee a[j] \simeq b[j]}$$

Fig. 3. Restricted $\Uparrow_r$ inference rule.

Theorem 11: The rules idx, $\Downarrow$, $\Uparrow_r$, $\mathsf{ext}_{\not\simeq}$ and ext_r are sound, terminating and complete.

Proof: The proof is similar to the proof of Theorem 6, but we use the completion procedure described above before defining $M^\lambda(a)$. ■

V. Combinatory Array Logic

In this section, we consider the extended array theory T_{CAL} with signature Σ_{CAL}. T_{CAL} contains two new families of combinators: the *constant-value* array combinators, and the *map* combinators. For each sort $(\sigma \Rightarrow \tau)$, Σ^F_{CAL} contains the function symbol $K{:}\,\tau \to (\sigma \Rightarrow \tau)$. For each function symbol $f{:}\,(\tau_1, \ldots, \tau_k) \to \tau$, interpreted or not, and sort σ, Σ^F_{CAL} contains the function symbol $\mathit{map}_f{:}\,((\sigma \Rightarrow \tau_1), \ldots, (\sigma \Rightarrow \tau_k)) \to (\sigma \Rightarrow \tau)$. We say map_f is the *pointwise array extension* of f. The following scheme axiomatizes the new combinators:

$$\forall v{:}\,\tau,\ i{:}\,\sigma\,.\ K(v)[i] \simeq v$$
$$\forall a_1{:}\,(\sigma \Rightarrow \tau_1), \ldots, a_k{:}\,(\sigma \Rightarrow \tau_k),\ i{:}\,\sigma\,.$$
$$\mathit{map}_f(a_1, \ldots, a_k)[i] \simeq f(a_1[i], \ldots, a_k[i])$$

Similarly, given a predicate symbol $p{:}\,(\tau_1, \ldots, \tau_k)$, we define the pointwise extension combinator map_p for predicates as:

$$\forall b_1{:}\,(\sigma \Rightarrow \tau_1), \ldots, b_k{:}\,(\sigma \Rightarrow \tau_k),\ i{:}\,\sigma\,.$$
$$(\neg p(b_1[i], \ldots, b_k[i]) \vee \mathit{map}_p(b_1, \ldots, b_k)[i] \simeq \top) \wedge$$
$$(p(b_1[i], \ldots, b_k[i]) \vee \mathit{map}_p(b_1, \ldots, b_k)[i] \simeq \bot)$$

Due to space limitations, we only discuss combinators of the form map_f. The extension to map_p is straight-forward.

From now on, for each sort τ, we assume the core theory contains the if-then-else operator $\mathit{ite}{:}\,(\mathsf{bool}, \tau, \tau) \to \tau$. The following scheme axiomatizes ite:

$$\forall x_1, x_2{:}\,\tau . \mathit{ite}(\top, x_1, x_2) \simeq x_1 \ \wedge \mathit{ite}(\bot, x_1, x_2) \simeq x_2$$

A. Versatility

The extended combinators allow to easily express some functions over sets and bags. The idea is to represent a set of elements of sort σ as an array of sort $(\sigma \Rightarrow \mathsf{bool})$. We list a few of these below.

$\emptyset$	$= K(\bot)$	$\overline{a}$	$= \mathit{map}_{ite}(a, K(\bot), K(\top))$
$\{v\}$	$= \mathit{store}(K(\bot), v, \top)$	$a \cup b$	$= \mathit{map}_{ite}(a, K(\top), b)$
$v \in a$	$= a[v]$	$a \cap b$	$= \mathit{map}_{ite}(a, b, K(\bot))$

Similarly, a bag (or multi-set) of elements of sort σ can be encoded as an array of sort $(\sigma \Rightarrow \mathsf{int})$. Then, the empty bag is encoded as $K(0)$, the set of elements in a bag a is $\mathit{map}_{>}(a, K(0))$, the multi-set extension of a set a is $\mathit{map}_{ite}(a, K(1), K(0))$, and the join operation $a \uplus b$ on bags is encoded as $\mathit{map}_{+}(a, b)$. On the other hand, the cardinality of a set/bag cannot be expressed.

$$\mathsf{K}\Downarrow \frac{a \equiv K(v), \quad w \equiv a'[j], \quad a \sim a'}{a[j] \simeq v}$$

$$\mathsf{map}\Downarrow \frac{a \equiv \mathit{map}_f(b_1, \ldots, b_n), \quad w \equiv a'[j], \quad a \sim a'}{a[j] \simeq f(b_1[j], \ldots, b_n[j])}$$

$$\mathsf{map}\Uparrow \frac{\begin{array}{c} a \equiv \mathit{map}_f(b_1, \ldots, b_n), \quad w \equiv b'_k[j], \\ b_k \sim b'_k, \text{ for some } \quad k \in \{1, \ldots, n\} \end{array}}{a[j] \simeq f(b_1[j], \ldots, b_n[j])}$$

$$\epsilon_{\not\simeq} \frac{v \equiv a[i], \quad i\colon\sigma, \quad i \text{ is not } \epsilon_\sigma}{\epsilon_\sigma \not\simeq i} \qquad \epsilon\delta \frac{a\colon(\sigma \Rightarrow \tau)}{a[\epsilon_\sigma] \simeq \delta_a}$$

Fig. 4. Extended combinators inference rules.

Notice also that $\mathit{store}(a, i, v) = \mathit{map}_{\mathit{ite}}(\{i\}, K(v), a)$, so we could use store as a derived combinator if we instead assume ite and the singleton combinator.

B. Extended Inference Rules

Fig. 4 contains the inference rules for the new combinators. In the proof of Theorem 6, for every array constant a, we defined $M^\lambda(a)(\iota) = \delta_\sigma$ if $\iota \notin \mathsf{index}(a)$, where δ_σ is an arbitrary value of $|M^\lambda|_\sigma$. That is, δ_σ is the *default value* of every array constant in Γ. This simple construction is not possible when combinators K and map_f are used, because they constrain the default value of array constants. Given an array constant a, we use the fresh constant $\delta_a\colon\sigma$ to denote the *default value* for array a. The rule $\epsilon\delta$ exposes the default value δ_a (of an array constant a) by accessing a at an index ϵ_σ. We have a fresh constant ϵ_σ for each sort σ. The rule $\epsilon_{\not\simeq}$ enforces that ϵ_σ is different from any other index i of sort σ. Of course, in general, the rule $\epsilon_{\not\simeq}$ is not sound if the interpretation of sort σ is finite. The following example illustrates the problem.

Example 12: Let i be a constant of sort σ. Then, the formula $\mathit{store}(K(v_1), i_1, w_1) \simeq K(v_2),\ v_1 \not\simeq v_2$ is satisfiable in a structure where the interpretation of σ has only one element. On the other hand, a procedure based on the inference rules in Fig. 1 and 4 will return unsatisfiable.

Theorem 13: Considering the simplifying assumption that the intended interpretation of every index sort σ is infinite, then the rules idx, $\Downarrow$, $\Uparrow$, $\mathsf{ext}_{\not\simeq}$, ext_r, $\mathsf{K}\Downarrow$, $\mathsf{map}\Downarrow$, $\mathsf{map}\Uparrow$, $\epsilon_{\not\simeq}$ and $\epsilon\delta$ are sound, terminating and complete.

Proof: The restricted version of the rule $\Uparrow$ is not considered here. We consider this optimization, in the context of extended combinators, in Section V-D. The proof is similar to the proof of Theorem 6, but the construction of M^λ is slightly different. We define the map $\mathsf{graph}(a)$ as before, but we define $M^\lambda(a)$ as:

$$M^\lambda(a)(\iota) = \begin{cases} \nu & \text{if } (\iota, \nu) \in \mathsf{graph}(a), \\ M^\lambda(\delta_a) & \text{otherwise.} \end{cases}$$

Now, we show that M^λ satisfies all definitions of the form $a \equiv \mathit{store}(b, i, v)$, $a \equiv K(v)$ and $a \equiv \mathit{map}_f(b_1, \ldots, b_k)$. Let $\mathsf{index}(a)$ be the set $\{\iota \mid (\iota, \nu) \in \mathsf{graph}(a)\}$. First, let us consider definitions of the form $a \equiv \mathit{store}(b, i, v)$, The rule idx guarantees that $M^\lambda(a)(M^\lambda(i)) = M^\lambda(v)$. The rules $\Uparrow$ and $\Downarrow$ guarantee that $\mathsf{index}(a) = \mathsf{index}(b) \cup \{M^\lambda(i)\}$, and $M^\lambda(a)(\iota) = M^\lambda(b)(\iota)$ for every $\iota \in \mathsf{index}(a) \setminus \{M^\lambda(i)\}$. Now, we just need to show that for every $\kappa \notin \mathsf{index}(a)$, $M^\lambda(a)(\kappa) = M^\lambda(b)(\kappa)$. This equality is a consequence of the following observations:

$$\begin{array}{lll} & M^\lambda(a)(\kappa) & \\ = & M^\lambda(\delta_a) & \text{(by def. of } M^\lambda) \\ = & M^\lambda(a)(M^\lambda(\epsilon_\sigma)) & \text{(by rule } \epsilon\delta) \\ = & M^\lambda(b)(M^\lambda(\epsilon_\sigma)) & \text{(by rule } \epsilon_{\not\simeq},\ M^\lambda(\epsilon_\sigma) \neq M^\lambda(i)) \\ = & M^\lambda(\delta_b) & \text{(by rule } \epsilon\delta) \\ = & M^\lambda(b)(\kappa) & \text{(by def. of } M^\lambda) \end{array}$$

For definitions of the form $a \equiv K(v)$, by rules $\mathsf{K}\Downarrow$ and $\epsilon\delta$, it is easy to see that $M^\lambda(a)(\iota) = M^\lambda(v)$ for all $\iota \in |M^\lambda|_\sigma$. Finally, we consider definitions of the form $a \equiv \mathit{map}_f(b_1, \ldots, b_k)$. The rule $\mathsf{map}\Downarrow$ guarantees that for all $\iota \in \mathsf{index}(a)$ the map_f axiom holds. The rule $\mathsf{map}\Uparrow$ guarantees that $\mathsf{index}(b_1) \cup \ldots \cup \mathsf{index}(b_k) \subseteq \mathsf{index}(a)$. Hence, if $\kappa \notin \mathsf{index}(a)$, then $\kappa \notin \mathsf{index}(b_1) \cup \ldots \cup \mathsf{index}(b_k)$. Then, by rule $\epsilon\delta$ and the definition of M^λ, we have $M^\lambda(a)(\kappa) = M^\lambda(a)(M^\lambda(\epsilon_\sigma))$, and for each $i \in \{1, \ldots, k\}$, $M^\lambda(b_i)(\kappa) = M^\lambda(b_i)(M^\lambda(\epsilon_\sigma))$. Since $M^\lambda(\epsilon_\sigma) \in \mathsf{index}(a)$, the map_f axiom is also satisfied for all $\kappa \notin \mathsf{index}(a)$. ∎

$$\mathsf{blast} \frac{a\colon(\sigma \Rightarrow \tau), \quad \mathsf{size}(\sigma) = k}{a[\sigma_1] \simeq \delta_{a,1}, \ \ldots, \ a[\sigma_k] \simeq \delta_{a,k}}$$

Fig. 5. Blasting inference rule.

A procedure using rule $\epsilon_{\not\simeq}$ may track how many times this rule was used. Let $\mathsf{num}(\sigma)$ be the number of times rule $\epsilon_{\not\simeq}$ was applied to ϵ_σ for indices of sort σ. Now, assume the size $\mathsf{size}(\sigma)$ of the intended interpretation of a sort σ is known. Then, it is sound to apply $\epsilon_{\not\simeq}$ when $\mathsf{num}(\sigma) < \mathsf{size}(\sigma)$. In practice, if $\mathsf{size}(\sigma)$ is very big (e.g., σ is the sort of bit-vectors of size 32), then it is "sound" to apply rule $\epsilon_{\not\simeq}$. If $\mathsf{size}(\sigma)$ is small and $\mathsf{num}(\sigma) \geq \mathsf{size}(\sigma)$, then instead of using rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$ a procedure may use the rule blast described in Fig. 5. In rule blast, each $\delta_{a,i}$ is a fresh constant, and σ_i is an interpreted constant that is a name for the i-th value in the intended interpretation of σ. For example, if σ is the sort bool, then $\mathsf{size}(\sigma) = 2$, σ_1 is $\top$ and σ_2 is $\bot$.

Finally, we consider the case where $\mathsf{size}(\sigma)$ is not known (e.g., σ is an uninterpreted sort). Then, given a formula φ, if a procedure using rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$ returns unsatisfiable, then we know that φ is unsatisfiable in any structure where the size of the interpretation of each index sort σ is greater than $\mathsf{num}(\sigma)$. The value $\mathsf{num}(\sigma)$ gives us a bound on the size of any interpretation of σ. A complete and sound procedure can be implemented using these bounds and the rule blast. This is essentially the equivalent of a finite model finding procedure. In general, this procedure is quite expensive. For example, if F contains n uninterpreted sorts $\sigma_1, \ldots, \sigma_n$, then we have to consider $\mathsf{num}(\sigma_1) \times \ldots \times \mathsf{num}(\sigma_n)$ additional satisfiability subproblems. If all of them are unsatisfiable, then F is indeed unsatisfiable.

$$\mathsf{U}\delta\ \frac{a \equiv \mathit{store}(b,i,v)}{\delta(a) \simeq \delta(b)} \qquad \mathsf{K}\delta\ \frac{a \equiv K(v)}{\delta(a) \simeq v}$$

$$\mathsf{map}\delta\ \frac{a \equiv \mathit{map}_f(b_1,\ldots,b_n)}{\delta(a) \simeq f(\delta(b_1),\ldots,\delta(b_n))}$$

Fig. 6. Default value inference rules.

The constants δ_a enable another filter for the rule ext_r. The idea is to only apply ext_r when $\delta_a \sim \delta_b$. The basic observation is that $M^\lambda(a) \neq M^\lambda(b)$ if $M^\lambda(\delta_a) \neq M^\lambda(\delta_b)$, when the index sort σ has a sufficiently big interpretation. This observation is equivalent to the filter used in [13]. The filter is not sound if $\mathsf{size}(\sigma)$ is finite (and small) because we might have $\mathsf{index}(a) = \mathsf{index}(b) = |M^\lambda|_\sigma$ and there is no $\iota \in |M^\lambda|_\sigma$ s.t. $M^\lambda(a)(\iota) = M^\lambda(\delta_a)$ and $M^\lambda(b)(\iota) = M^\lambda(\delta_b)$.

C. Default Value Propagation

In this section, we use the simplifying assumption that every index sort is infinite. A corollary of Theorem 8 is that if a formula φ is satisfiable in the extended array theory, then it is satisfied in a structure M^λ where for every array constant a there is a value $M^\lambda(\delta_a)$ s.t. there is only a finite number of indices ι such that $M^\lambda(a)(\iota) \neq M^\lambda(\delta_a)$. Thus, we say that every array constant a has a *default value*. This observation suggests an alternative to rules $\epsilon_{\not\simeq}$ and $\epsilon\delta$. The idea is to propagate constraints about the default value of each array constant a. We use distinguished function symbols δ, and encode the default value of a as the term $\delta(a)$. Fig. 6 contains the inference rules for propagating default values.

The distinguished functions may be used to encode properties of arrays. If we want to force a set b to be finite, we can assert $\delta(b) = \bot$ as part of the formula checked for satisfiability.

D. Restricted $\Uparrow_r$ and $\mathsf{map}\Uparrow_r$ for Extended Combinators

Now, we consider the problem of restricting the inference rules $\Uparrow$ and $\mathsf{map}\Uparrow$. The construction used in Theorem 11 does not work. For example, the extended array theory has combinators that take many array arguments. Given a definition of the form $a \equiv \mathsf{C}(\ldots,b,\ldots)$, where C is an arbitrary combinator, the basic idea was to use (a subset of) $\mathsf{graph}(b)$ to complete the map $\mathsf{graph}(a)$, when b is linear. However, if a combinator contains many array arguments $b_1,\ldots,b_k$, then we may have $\iota \in \mathsf{index}(b_i)$, but $\iota \notin \mathsf{index}(b_j)$ for some i and j in $\{1,\ldots,k\}$. It is incorrect to assume $M^\lambda(b_j)(\iota) = M^\lambda(b_j)(M^\lambda(\epsilon_\sigma))$, because $M^\lambda(b_j)$ may not have been defined yet when constructing $M^\lambda(a)$. The following example illustrates this problem.

Example 14: Let a, b and c be arrays $(\sigma \Rightarrow \mathsf{bool})$. Let us assume we are using a restricted version of $\mathsf{map}\Uparrow$ similar to $\Uparrow_r$. Now, consider the following formula:

$$a \simeq \mathit{map}_{ite}(a,b,c) \ \wedge\ b[j] \simeq \bot \ \wedge\ c[j] \simeq \top$$

The constant a is a linear parent [1], because its equivalence class contains only one combinator $\mathit{map}_{ite}(a,b,c)$. Therefore, the restricted version of $\mathsf{map}\Uparrow$ does not instantiate the *map* axiom, and the unsatisfiability is not detected. Note that if $a[j]$ is $\top$, then we have an inconsistency because $b[j] \simeq \bot$. Similarly, if $a[j]$ is $\bot$, then we also have an inconsistency because $c[j] \simeq \top$.

The example above suggests a simple solution based on a total order $\prec$ on constants compatible with the order $\sqsubset$ on sorts. By compatible, we mean that if $v\colon\sigma$, $w\colon\tau$ and $\sigma \sqsubset \tau$ implies $v \prec w$. The order $\prec$ allows us to define a notion of stratification that complements the definition of linearity defined in Section IV.IV-C. We use $a \preceq b$ to denote $a \prec b$ or $a = b$.

Definition 15 (Linear Stratification): Given a state Γ, the set $\mathsf{non\text{-}linear\text{-}stratified}$ of *non-linear-stratified constants* is the least set such that:

1) $a_1 \equiv \mathsf{C}(\ldots)$, $a_2 \equiv \mathsf{C}'(\ldots)$, a_1 is not a_2 and $a_1 \sim a_2$ implies $\{a_1,a_2\} \subseteq \mathsf{non\text{-}linear\text{-}stratified}$,
2) $a \equiv \mathsf{C}(\ldots,b,\ldots)$, $b\colon(\sigma \Rightarrow \tau)$, $b \sim b'$ and $a \preceq b'$ implies $a \in \mathsf{non\text{-}linear\text{-}stratified}$,
3) $a \equiv \mathsf{C}(\ldots,b,\ldots)$, $b\colon(\sigma \Rightarrow \tau)$ and $a \in \mathsf{non\text{-}linear\text{-}stratified}$ implies $b \in \mathsf{non\text{-}linear\text{-}stratified}$,
4) $a \in \mathsf{non\text{-}linear\text{-}stratified}$ and $a \sim b$ implies $b \in \mathsf{non\text{-}linear\text{-}stratified}$.

where, C and C' are arbitrary combinators. We say a is *linear-stratified* if $a \notin \mathsf{non\text{-}linear\text{-}stratified}$.

Now, we restrict the application of the rules $\Uparrow$ and $\mathsf{map}\Uparrow$ using the $\mathsf{non\text{-}linear\text{-}stratified}$ instead of $\mathsf{non\text{-}linear}$. Let $\Uparrow_r$ and $\mathsf{map}\Uparrow_r$ be the restricted version of these rules. If a is linear stratified and $a \equiv \mathsf{C}(\ldots,b,\ldots)$, then we can assume that $M^\lambda(b)$ was already defined when defining $M^\lambda(a)$.

Theorem 16: Considering the simplifying assumption that the intended interpretation of every index sort σ is infinite, then the rules idx, $\Downarrow$, $\Uparrow_r$, $\mathsf{ext}_{\not\simeq}$, ext_r, $\mathsf{K}\Downarrow$, $\mathsf{map}\Downarrow$, $\mathsf{map}\Uparrow_r$, $\epsilon_{\not\simeq}$ and $\epsilon\delta$ are sound, terminating and complete.

Proof: The proof is similar to the proof of Theorem 8. Let $\overline{a}$ be the greatest constant, with respect to the order $\prec$, in the equivalence class containing a. Now, we define $M^\lambda(\overline{a})$ only after all constants $\overline{b} \prec \overline{a}$ were already defined. Similarly, if $f\colon(\sigma_1,\ldots,\sigma_k) \to \tau$ is uninterpreted, then we define $M^\lambda(f)$ after all all constants of sorts $\sigma_1,\ldots,\sigma_k$ and τ were already defined. If $\overline{a}$ is linear-stratified and $a \equiv \mathit{map}_f(b_1,\ldots,b_k)$, then we define $M^\lambda(\overline{a})$ as:

$$M^\lambda(\overline{a})(\iota) = \begin{cases} \nu \text{ if } (\iota,\nu) \in \mathsf{graph}(\overline{a}), \\ M^\lambda(f)(M^\lambda(b_1)(\iota),\ldots,M^\lambda(b_k)(\iota)), \text{ otherwise.} \end{cases}$$

The construction for definitions of the form $a \equiv \mathit{store}(b,i,v)$ is similar. If $\overline{a}$ is not linear-stratified, then we use the same construction used in the proof of Theorem 8. After defining $M^\lambda(\overline{a})$, we make $M^\lambda(a) = M^\lambda(\overline{a})$ for every a in the equivalence class of $\overline{a}$. ■

The proof of Theorem 4 established that the reduction from $T_A \oplus T_{\mathsf{core}}$ to T_{core} required at most a cubic number of new terms. The reduction of $T_{\mathsf{CAL}} \oplus T_{\mathsf{core}}$ to T_{core} can also be bounded by a polynomial number of new terms, using a similar argument, and:

[1] defined in the technical report

Theorem 17: If the satisfiability problem for T_{core} is NP-complete, then the satisfiability problem for $T_{\mathsf{core}} \oplus T_{\mathsf{CAL}}$ is also NP-complete.

VI. Experimental Results

First, let us describe implementation details that are relevant for reproducing our results. Hence, we describe how the proposed inference rules were implemented in the SMT solver Z3. The rule idx is applied whenever a definition of the form $a \equiv \mathit{store}(b, i, v)$ is created. The rules $\epsilon_{\not\approx}$ and $\epsilon\delta$ are only used when the input formula contains the combinators K and map_f. In this case, $\epsilon\delta$ is applied when an array constant is created, and $\epsilon_{\not\approx}$ is delayed. Actually, we use *model-based instantiation* for guiding the application of the rule $\epsilon_{\not\approx}$. The idea is to build a candidate model M^λ without even using $\epsilon_{\not\approx}$, if in this model $M^\lambda(\epsilon_\sigma) \neq M^\lambda(i)$ for every i ($\neq \epsilon_\sigma$) used as an index, then we have a valid model. Otherwise, we expand $\epsilon_{\not\approx}$ and continue. The rule $\mathsf{ext}_{\not\approx}$ is applied when p is assigned to false, and the application of ext_r is delayed. Before applying ext_r we build the set already-diseq. We use a simple indexing technique, called *use-lists*, for applying the remaining rules. Given a definition $a \equiv \mathsf{C}(\ldots, b, \ldots)$, we say a is a parent of b. The use-list data-structure stores the parents of each array constant a. Recall that we use an union-find data-structure for implementing equivalence classes. Then, whenever the union operation is performed we use the use-lists to find new matches for the remaining inference rules. The sets foreign, non-linear and non-linear-stratified are implemented as mappings from equivalence class representatives to Booleans.

Fig. 7 contains two scatter-graphs [2] comparing the performance of Z3 with and without the rule $\Uparrow_r$ in all 2244 benchmarks in the QF_AUFLIA division of SMT-LIB. Each point on the plots represents a benchmark. On each plot the x-axis is the CPU time, in seconds, taken by Z3 using $\Uparrow_r$, and y-axis in the first graph is for Z3 using $\Uparrow$, and in the second graph is for Z3 delaying the application of $\Uparrow_r$. Points above the diagonal are then benchmarks where Z3 with $\Uparrow_r$ is faster. The scatter-graphs clearly show that rule $\Uparrow_r$ increases Z3's performance in hard instances. Note that delaying the application of $\Uparrow_r$ increases the performance in unsatisfiable benchmarks because most array constants are linear, and, consequently, this rule is not needed. However, in satisfiable instances the rule is eventually applied and performance degrades.

VII. Conclusion

We described efficient and simple decision procedures for the array theory and combinatory array logic. The new combinators admit a simple theory of sets and bags. The theory of sets has already been used in real applications at Microsoft (e.g., the program exploration tool Pex and SpecExplorer). The decision procedure is presented as a set of inference rules on top of a core solver which provides basic capabilities. We also described the implementation techniques used to efficiently implement these inference rules. Moreover, in our approach the index domain of an array can be finite (e.g., bit-vectors). We also described several filters for minimizing the number of times an inference rule needs to be applied while retaining completeness. In particular, our experiments show that rule $\Uparrow_r$ improves the performance of Z3.

[2] Data available at http://research.microsoft.com/~leonardo/fmcad09

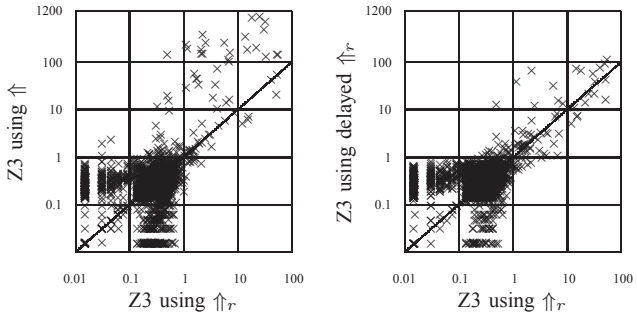


Fig. 7. Z3 on QF_AUFLIA Benchmarks

References

[1] McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. (1962) 21–28

[2] Suzuki, N., Jefferson, D.: Verification Decidability of Presburger Array Programs. J. ACM **27** (1980) 191–205

[3] Jaffar, J.: Presburger Arithmetic With Array Segments. Inf. Process. Lett. **12** (1981) 79–82

[4] Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: CAV. (1994)

[5] Manolios, P., Srinivasan, S.K., Vroon, D.: Automatic memory reductions for RTL model verification. In: ICCAD. (2006)

[6] Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS. (2001)

[7] Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. **10** (2009)

[8] Lynch, C., Morawska, B.: Automatic decidability. In: LICS. (2002)

[9] Brummayer, R., Biere, A.: Lemmas on Demand for the Extensional Theory of Arrays. In: SMT. (2008)

[10] Kapur, D., Zarba, C.G.: A Reduction Approach to Decision Procedures. Technical Report TR-CS-1005-44, University of New Mexico (2005)

[11] Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: CAV. Volume 4144 of LNCS. (2006)

[12] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. (2008)

[13] Amit Goel and Sava Krstic and Alexander Fuchs: Deciding Array Formula with Frugal Axiom Instantiation. In: SMT. (2008)

[14] Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV. (2007) 519–531

[15] Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodriguez-Carbonell, E., Rubio, A.: A Write-Based Solver for SAT Modulo the Theory of Arrays. In: FMCAD. (2008) 1–8

[16] Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI. (2006) 427–442

[17] Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Ann. Math. Artif. Intell. **50** (2007) 231–254

[18] Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: FoSSaCS. (2008)

[19] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM **53** (2006)

[20] de Moura, L., Bjørner, N.: Model-based Theory Combination. In: SMT. (2007)

[21] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52** (2005) 365–473

Decision Diagrams for Linear Arithmetic

Sagar Chaki
SEI/CMU
chaki@sei.cmu.edu

Arie Gurfinkel
SEI/CMU
arie@sei.cmu.edu

Ofer Strichman
Technion
ofers@ie.technion.ac.il

***Abstract*—Boolean manipulation and existential quantification of numeric variables from linear arithmetic (LA) formulas is at the core of many program analysis and software model checking techniques (e.g., predicate abstraction). We present a new data structure, Linear Decision Diagrams (LDDs), to represent formulas in LA and its fragments, which has certain properties that make it efficient for such tasks. LDDs can be seen as an extension of Difference Decision Diagrams (DDDs) to full LA. Beyond this extension, we make three key contributions. First, we extend sifting-based dynamic variable ordering (DVO) from BDDs to LDDs. Second, we develop, implement, and evaluate several algorithms for existential quantification. Third, we implement LDDs inside CUDD, a state-of-the-art BDD package, and evaluate them on a large benchmark consisting of 850 functions derived from the source code of 25 open source programs. Overall, our experiments indicate that LDDs are an effective data structure for program analysis tasks.**

I. Introduction

Many program analysis problems – e.g., computation and application of predicate abstraction, image computation, function summarization – are ultimately reduced to manipulating propositional formulas over some theory. Typically, two types of operations are required: (i) Boolean (conjunction, negation, etc.) and (ii) existential quantification (henceforth QELIM). For example, the predicate abstraction of a transition relation R is computed as

$$\exists \vec{x}, \vec{x'} \,.\, R(\vec{x}, \vec{x'}) \wedge \left(\bigwedge_{i=1}^{n} v_i \Leftrightarrow p_i(\vec{x}) \right) \wedge \left(\bigwedge_{i=1}^{n} v'_i \Leftrightarrow p_i(\vec{x'}) \right) , \quad (1)$$

where $\vec{x}, \vec{x'}$ are current- and next-sate program variables, $\vec{v}, \vec{v'}$ are current- and next-state propositional abstract variables, and p_i is the definition of the i-th abstract variable in terms of $\vec{x}$. Thus, effective predicate abstraction – and program analysis tasks in general – requires a mechanism that combines space-efficient representation of formulas and fast QELIM.

This is challenging because QELIM algorithms require a formula in Disjunctive Normal Form (DNF), but DNF is a space-inefficient representation, e.g., relative to Conjunctive Normal Form (CNF). Common solutions to this issue follow one of three approaches: DNF, Abstract Syntax Tree (AST), and Decision Diagram (DD). In the DNF approach, a formula is represented by the set of terms of a DNF, e.g., a disjunctive invariant is represented by a set of octagons [15]. This approach is easy to implement, but does not scale to formulas with large DNFs. In the AST approach, a formula is represented as an AST (or a DAG) whose nodes correspond to variables, constants, and operators. This is the most space-efficient representation. However, converting an AST into DNF during QELIM – e.g., using a SMT-solver [12] – is expensive, in many cases exponential in the AST size. In the DD approach, a formula is represented by a DAG whose nodes are labeled by atomic terms. The DAG enables sharing of sub-expressions, and, at the same time, is easy to convert to DNF. This is the approach we explore in this paper by introducing and evaluating a new data structure, Linear Decision Diagram (LDD), for quantifier-free first-order linear arithmetic (LA) formulas.

LDDs extend DDDs – decision diagrams for difference logic – proposed by Møller et al. [16]. Predicates in difference logic are of the form $x - y \leq c$ or $x - y < c$ for variables x, y and a constant c, ranging over $\mathbb{Q}$ or $\mathbb{Z}$. The key idea of DDDs is to represent first-order quantifier-free difference logic formulas as BDDs with nodes labeled by atomic predicates, and to reduce redundancy by leveraging implications between those predicates. For example, in a DDD, a node labeled with $x - y \leq 10$ never appears as a high child of a node labeled with $x - y \leq 5$. For a fixed variable order, a DDD of a formula f is no larger than a BDD representing a propositional abstraction of f (i.e., f with all atomic terms replaced by propositional variables). An important feature of DDDs is a QELIM algorithm based on Fourier-Motzkin elimination [3]. The algorithm lends itself well to dynamic programming, thus, leveraging the DAG-structure of DDs.

DDDs have three main limitations as an instrument to aid program analysis: (a) difference logic is too restrictive for many program analysis tasks, (b) DDDs do not support dynamic variable ordering (DVO), and (c) there are no publicly available implementations of DDDs and no reports of their effectiveness in solving practical program analysis problems. Our solution via LDDs address all three limitations. Specifically, LDDs extend to full linear arithmetic, and support efficient algorithms for DVO and QELIM. Moreover, we implemented LDD within CUDD, a state-of-the-art BDD package. Finally, we have evaluated LDDs using a benchmark derived from open-source programs.

Extending the Boolean operations from DDDs to LDDs is straightforward. The key challenges are in supporting DVO and QELIM. The importance of DVO for DDs is well-known. This is especially true for LDDs since they add restrictions on the variable order. We show that the standard BDD DVO *cannot* be used "as is" for LDDs. Instead, we adapt Rudell's sifting algorithm [18], as implemented in CUDD. Our adaptation does not add any overhead over the BDD version of DVO. Our experiments confirm that DVO reduces the size of LDDs significantly.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

We develop two types of QELIM for LDDs. The *black-box* QELIM applies an external QELIM-solver to each path in the LDD. It is linear in the number of paths (bad), but is compatible with any off-the-shelf solver (good). In contrast, the *white-box* QELIM is a generalized and optimized variant of the DDD version. It recursively applies pairwise resolution to DD nodes, in the style of Fourier-Motzkin. In the worst case, this algorithm is exponential in the size of the diagram. However, our experiments indicate that it performs well in practice. Black-box and white-box QELIM have success rates of 4.47% and 98.94%, respectively, when solving our benchmark problems.

The basic white-box QELIM algorithm only eliminates one variable at a time – a fundamental limitation of Fourier-Motzkin. However, many applications of QELIM in program analysis require eliminating multiple variables. To address this, we present an elimination strategy that iterates, while there are variables to be eliminated, between (a) dropping constraints with variables that are not resolved on during Fourier-Motzkin, and (b) choosing and eliminating an existentially quantified variable which results in a minimum number of resolutions. This heuristic yields a speedup of over 5 times for QELIM in our benchmark.

In order to test the suitability of LDDs for solving practical program analysis programs, we have evaluated our implementation on a benchmark derived from open source programs. The benchmark consists of transition relations (in Static Single Assignment form) of 850 functions from 25 C programs. In each case, we measured the space required (in DD nodes) to represent the transition relation, and time to compute a forward image (i.e., strongest post-condition). The experimental results lead us to believe that LDDs are an effective representation for fragments of LA for program analysis tasks.

Related Work. DDs for theories other than propositional logic have been studied extensively since the early 90's, capitalizing on the great success of BDDs and numerous BDD optimizations. Among these, we do not cover DDs – such as Binary Moment Diagrams (BMDs) [5], Algebraic Decision Diagrams (ADDs) [2], and Boolean Expression Diagrams (BEDs) [1] – which are restricted to variables with finite domains. Thus, the most relevant structures to LDDs are those that label the nodes with linear predicates over the reals.

Groote and Tveretina [9] proposed decision diagrams for full first-order logic. Assuming an input formula is in Prenex normal form, they represent the negation of its quantification suffix with a DD, and prove a contradiction using Skolemization and standard strategies, e.g., applying unifiers.

Equational (EQ) BDDs [10] are aimed at deciding equalities with uninterpreted functions (EUF). In EQ-BDDs, nodes are labeled with predicates (equalities), and reduction rules enforce substitution according to a predefined order $\preceq$ between variables. For example, if $x \preceq y$, then y is substituted by x in high sub-DD of a node labeled $x = y$. Equational BDDs are semi-canonical – they reduce to $\mathbf{0}$ (or $\mathbf{1}$) if they represent a contradiction (or tautology), but are non-canonical otherwise.

Cavada et al. [6] propose a QELIM technique for LA that combines BDDs and SMT-solvers. They focus on predicate abstraction, and consider the problem of quantifying all numeric variables from LA formulas (see (1) for a template) over LA predicates and propositional variables. The Boolean structure of the formulas are encoded via BDDs. QELIM is done by recursively traversing the BDD, carrying, along each path, the set of linear predicates (i.e., the context) seen on it. At each recursive step, an SMT-solver is used to check whether the context is consistent. Paths with inconsistent context are removed. The use of the context precludes the use of dynamic programming. Thus, the algorithm is linear in the number of paths of the BDD. We call such an approach "black box" because it uses an external decision procedure as is. In Sec. IV, we present a similar "black box" algorithm for quantifying *some* or all numeric variables.

The most relevant prior work is on DDDs [16]. QELIM of a numeric variable from a DDD is based on Fourier-Motzkin. Although in the worst case this procedure is exponential in the size of the DD, in the best case it is the same as QELIM for BDDs. We describe it in more detail in Sec. IV, as it is the base for our improved method. *Clock Difference Diagrams* [13], is an alternative to DDDs that was developed independently around the same time. CDDs are based on DDs with arbitrary branching degree. QELIM is done in the black-box fashion by traversing all $\mathbf{1}$-paths. We are not aware of any work that has adapted DVO to either DDDs, CDDs, or EQ-BDDs. DDDs and CDDs are extensions of Interval Decision Diagrams (IDD) [19].

We define LDDs in the next section. In Sec. III and IV we discuss the problems of dynamic variable ordering and QELIM with such diagrams, respectively. Sec. V is dedicated to experimental results, and we conclude in Sec. VI.

II. LINEAR DECISION DIAGRAMS

We assume the reader is familiar with the basics of decision diagrams. A Linear Decision Diagram (LDD) is a data structure to represent and manipulate propositional formula over (a fragment of) linear arithmetic. Formally, they are BDDs with (a) nodes labeled by linear atomic predicates, and (b) satisfying ordering and local reduction constraints. In the rest of this paper, we use T to denote a fragment of linear arithmetic, unless mentioned otherwise. For a formula p, we write $\text{VARS}(p)$ to mean the set of variables in p.

A. Definitions

An LDD over a theory T is a directed acyclic graph with

- Two terminal nodes labeled with $\mathbf{0}$ and $\mathbf{1}$, respectively;
- Nonterminal nodes. Each nonterminal node u has two children, denoted by $H(u)$ and $L(u)$, and is labelled with a T-atom (i.e., an atomic predicate), denoted by $C(u)$.
- Edges $(u, H(u))$ and $(u, L(u))$ for every non-terminal node u.

We use $attr(u)$ to denote the triple $(C(u), H(u), L(u))$. An LDD with a root node u represents the formula $exp(u)$ over

T defined as follows: $exp(\mathbf{0})$ is FALSE, $exp(\mathbf{1})$ is TRUE, otherwise, $exp(u)$ is defined recursively as:

$$exp(u) = \text{ITE}(C(u), exp(H(u)), exp(L(u))) \ ,$$

where

$$\text{ITE}(a, b, c) = (a \wedge b) \vee (\neg a \wedge c) \ .$$

For simplicity, we don't distinguish between a node u and $\exp(u)$.

Example 1 *An example of an LDD for the formula*

$$(z - y \leq 0 \wedge x - y \leq 10) \vee (z - y > 0 \wedge x - y \leq 5)$$

is shown in Fig. 3(a). A different LDD for the same formula, owing to a different order, is shown in Fig. 3(b).

B. Requirements from the theory

A theory T over variables Var is *LDD-adequate*, or simply *adequate*, if for any given set of consistent T-atoms A_T over Var (e.g., $x < x$ is an inconsistent T-atom for T being linear arithmetic), the following functions can be provided:

- (*Negation*) $NEG : A_T \mapsto A_T$ such that $NEG(c) \Leftrightarrow \neg c$. In the following, we write $\neg c$ to mean $NEG(c)$.
- (*Normalization*) $N : A_T \mapsto A_T$ such that if $c \Leftrightarrow c'$ or $c \Leftrightarrow \neg c'$ then $N(c) = N(c')$. Note that $\forall c \in A_T . N(c) = N(\neg c)$. If $c = N(c)$, then c and $\neg c$ are represented by $\text{ITE}(c, \mathbf{1}, \mathbf{0})$ and $\text{ITE}(c, \mathbf{0}, \mathbf{1})$, respectively.
- (*Negation Check*) $isNEG : A_T \mapsto Bool$ such that $isNEG(c) \Leftrightarrow (c = \neg N(c))$.
- (*Implication*) $IMP : A_T \times A_T \mapsto Bool$ such that $IMP(c_1, c_2)$ iff $(c_1 \Rightarrow c_2)$.
- (*Resolution*) $\text{RSLV} : A_T \times Var \times A_T \mapsto A_T \cup \{\text{TRUE}\}$ such that $\text{RSLV}(c_1, x, c_2) = c_3$ iff $x \in \text{VARS}(c_1) \cap \text{VARS}(c_2)$, and $c_3 \Leftrightarrow \exists x . \ c_1 \wedge c_2$. We syntactically extend RSLV so its first argument is a set of T-atoms as follows: $\text{RSLV}(S, x, c) = \bigwedge_{s \in S} \text{RSLV}(s, x, c)$.

Example 2 *Let UTVPI be the quantifier-free first order theory of Unit Two Variables Per Inequality over the integers. The set of UTVPI-atoms is*

$$\{ax + by \leq k \mid x, y \in Var, a, b \in \{-1, 1\}, k \in \mathbb{Z}\} \ .$$

This theory is also known as Octagons *[15]. UTVPI is adequate, as shown below:*

- $NEG(ax + by \leq k) = -ax - by \leq -k - 1$.
- *Let* $<_V$ *be a total order on* Var *and* $c = ax + by \leq k$. *Then,* $N(c)$ *is (i)* $N(by + ax \leq k)$ *if* $y <_V x$, *(ii)* c *if* $x <_V y \wedge a > 0$, *and (iii)* $NEG(c)$ *otherwise.*
- $IMP(c_1, c_2) = \text{TRUE}$ *iff* $c_1 = ax + by \leq k$ *and* $c_2 = ax + by \leq k'$ *and* $k \leq k'$.
- $\text{RSLV}(c_1, x, c_2)$ *is* TRUE *if* x *does not appear in opposite phases in* c_1 *and* c_2. *Otherwise, it is the result of resolution on* x *between* c_1 *and* c_2. *For example,* $\text{RSLV}(x - y \leq 5, x, z - x \leq 2) = z - y \leq 7$.

Other adequate theories include: intervals, whose atoms are $\{x \leq k \mid x \in Var, k \in \mathbb{Z}\}$; difference logic, whose atoms are $\{x - y \leq k \mid x, y \in Var, k \in \mathbb{Z}\}$; and linear arithmetic over the reals. We use UTVPI for illustrations in this paper and in all of our experiments.

```
1: function MK (A_T c, LDD f, LDD g)
2:     if (IMP(c, C(f))) then f ← H(f)
3:     if (f = g) then return g
4:     if (IMP(c, C(g)) ∧ f = H(g)) then return g
5:     return BDDNODE(c, f, g)
```

Fig. 1. MK: Building an LDD.

C. Variable ordering

For LDDs, "T-atoms ordering" replaces the traditional BDD "variable ordering". Ordering between the T-atoms facilitates the construction of reduced diagrams. Let $\leq_{IMP}$ be the partial order on A_T induced by IMP: $c_1 \leq_{IMP} c_2 \Leftrightarrow IMP(c_1, c_2)$. A *$T$-atoms ordering* is any total order $\leq_T$ that extends $\leq_{IMP}$ to a total order on A_T. An LDD u is *ordered* w.r.t. $\leq_T$ iff for every node v reachable from u, $C(v) \leq_T C(H(v))$ and $C(v) \leq_T C(L(v))$. An LDD u is *well-ordered* if it is ordered with respect to some T-atoms ordering $\leq_T$. Both of the LDDs in Fig. 3 are well-ordered.

D. Local Reductions

An LDD is locally reduced iff the following five conditions hold on every internal node u and v.

1) *No duplicate nodes.* $attr(u) = attr(v) \Rightarrow u = v$.
2) *No redundant nodes.* $L(v) \neq H(v)$.
3) *Normalized labels.* $C(v) = N(C(v))$.
4) *Imply high.* $\neg IMP(C(v), C(H(v)))$.
5) *Imply low.* $IMP(C(v), C(L(v))) \Rightarrow H(v) \neq H(L(v))$.

For example, the LDD in Fig. 3(a) is reduced, but the LDD in Fig. 3(b) is not. From here on, we write LDD to mean Reduced Ordered LDD (ROLDD). The function $\text{MK}(c, f, g)$, shown in Fig. 1, constructs an ROLDD for $\text{ITE}(c, f, g)$, where (i) c is a normalized constraint, (ii) f and g are ROLDDs s.t. $f \neq g$, and (iii) $c \leq_T C(f)$ and $c \leq_T C(g)$. Lines 2–4 ensure that the result is reduced, and line 5 returns a unique diagram node representing the ITE. The proof of correctness of MK is based on the following two reduction rules to enforce *Imply high* and *Imply low*, respectively:

$$\frac{\text{ITE}(x, \text{ITE}(y, h, l), z) \quad IMP(x, y)}{\text{ITE}(x, h, z)} \ ,$$

$$\frac{\text{ITE}(x, y, \text{ITE}(z, h, l)) \quad IMP(x, z) \quad y \Leftrightarrow h}{\text{ITE}(z, h, l)} \ .$$

E. Basic LDD Operations

For any symmetric operator op that distributes over ITE, the function APPLY (Op op, LDD f, LDD g) constructs the LDD for $f \ op \ g$. It is similar to the equivalent BDD operation. It is based on the following transformations (if multiple rules apply, the earliest is selected) and their symmetric versions obtained by swapping the arguments of op:

$$\frac{\text{ITE}(x, y, z) \ op \ \text{ITE}(u, v, w) \quad x \Leftrightarrow u}{\text{ITE}(x, y \ op \ v, z \ op \ w)} \ ,$$

1: **function** SWAPINPLACE (i, j)
2: replace every BDD node $F : (i, H, L)$ with (j, G_1, G_0) where
3: F_{10}, F_{11} are the $\neg j$ and j cofactors of H
4: F_{00}, F_{01} are the $\neg j$ and j cofactors of L
5: $G_1 \leftarrow (F_{11} = F_{01})\ ?\ F_{11} : (i, F_{11}, F_{01})$
6: $G_0 \leftarrow (F_{00} = F_{10})\ ?\ F_{00} : (i, F_{10}, F_{00})$

Fig. 2. Swapping adjacent labels in a BDD.

$$\frac{\text{ITE}(x, y, z)\ op\ \text{ITE}(u, v, w) \quad IMP(x, u)}{\text{ITE}(x, y\ op\ v, z\ op\ \text{ITE}(u, v, w))}\ , \text{ and}$$

$$\frac{\text{ITE}(x, y, z)\ op\ \text{ITE}(u, v, w) \quad x \leq_T u}{\text{ITE}(x, y\ op\ \text{ITE}(u, v, w), z\ op\ \text{ITE}(u, v, w))}\ .$$

LDD conjunction (AND) and LDD disjunction (OR) are implemented via APPLY. LDD negation (NOT) is implemented as in BDDs. The implementation of ternary LDD if-then-else (ITE) is similar to APPLY.

In summary, extending Boolean operations from BDDs to LDDs is straightforward. In the next two sections, we show how to extend dynamic variable ordering and QELIM, which are more challenging.

III. DYNAMIC VARIABLE ORDERING

It is well known that the variable order of a decision diagram has a crucial effect on its size. Both finding the best order and deciding whether an order is optimal are NP-hard [4]. A lot of BDD research (e.g., [18], [8], [17]) has been dedicated to heuristics for finding a good variable order. In this section, we show how to adapt the sifting heuristic of Rudell [18], as implemented in CUDD, to LDDs. The exact details of the heuristic are beyond the scope of this paper. We only focus on the parts that we changed.

Sifting heuristic is based on trial-and-error. Each node label (i.e., a BDD variable) is moved up and down in the order by swapping the order of two adjacent labels. The best position is recorded and restored at the end. There are several factors that make sifting very efficient: a set of Boolean functions is represented as one multi-rooted DAG; a *unique table* is used to locate nodes in the DAG in constant time; the table is partitioned into subtables, one per label, to locate all nodes with a given label in O(1); and, finally, swapping adjacent labels i and j in all diagrams in the unique table is done in time linear in the number of nodes labeled with i or j.

Only the swapping algorithm needs to be adapted for LDDs. Swapping adjacent labels amounts to reordering diagrams in the unique table. In CUDD, this operation is called SWAPINPLACE. We first show how SWAPINPLACE works for BDDs, and then, that it does not work for LDDs.

Pseudocode for SWAPINPLACE is shown in Fig. 2. Note that the swapping is done *in place* – any edge that was pointing to a node labeled with i before the swap points to the same node, but now labeled with j after the swap. Furthermore, SWAPINPLACE maintains an invariant that every diagram in the unique table is well-ordered and reduced.

SWAPINPLACE does not work for LDDs. Consider an LDD shown in Fig. 3(a). Completely unrestricted swapping of adjacent labels conflicts with LDD well-orderedness constraints.

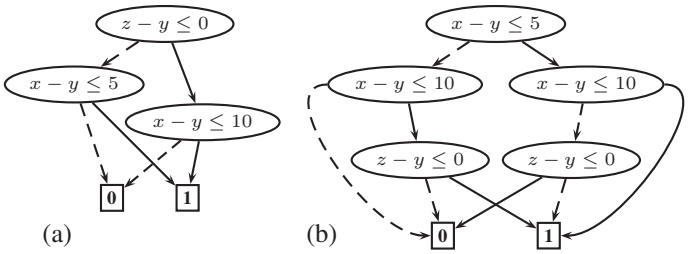


Fig. 3. (a) An ROLDD ordered by $z - y \leq 0$,$x - y \leq 5$, $x - y \leq 10$; (b) An OLDD ordered by $x - y \leq 5$, $x - y \leq$, $z - y \leq 0$.

1: **function** GROUPMOVE $(X \subseteq A_T\ ,\ Y \subseteq A_T)$
2: **for** $(i = |X|; i \geq 1; i \leftarrow i - 1)$ **do**
3: **for** $(j = 1; j \leq |Y|; j \leftarrow j + 1)$ **do**
4: LDDSWAPINPLACE(x_i, y_j)

Fig. 4. Swapping two adjacent sets of labels.

Say, we swap $z - y \leq 0$ and $x - y \leq 5$. Then, the result is not well-ordered since a node labeled $z - y \leq 0$ appears between the nodes labeled with $x - y \leq 5$ and $x - y \leq 10$. Thus, we must swap groups of variables at a time. Say, we further swap $z - y \leq 0$ with $x - y \leq 10$. The result is shown in Fig. 3(b). This LDD is well-ordered, but it is not reduced: the *Imply high* rule (see Sec. II-D) is violated. It is easy to construct an example where the *Imply low* rule is violated as well.

To overcome the problems shown above, we propose a new algorithm, called GROUPMOVE. GROUPMOVE takes two ordered sets of T-atoms $X = \{x_1, \ldots, x_{|X|}\}$ and $Y = \{y_1, \ldots, y_{|Y|}\}$, and swaps them in the variable order. The order before GROUPMOVE is $x_1, \ldots, x_{|X|}, y_1, \ldots, y_{|Y|}$, and after GROUPMOVE is $y_1, \ldots, y_{|Y|}, x_1, \ldots, x_{|X|}$. This is done with $|X| \times |Y|$ calls to the helper function LDDSWAPINPLACE. Intuitively, GROUPMOVE sifts each atom in the top class, one at a time, through the bottom class.

Function LDDSWAPINPLACE (shown in Fig. 5) is a variant of the original SWAPINPLACE that also maintains LDD reductions. Lines 7 and 10 ensure that any newly constructed LDD has no redundant nodes and satisfies *Imply low* rule. Lines 5 and 6 establish a stronger variant of *Imply high*: if a node v is reachable from a node u through $H(u)$, then $C(u)$ does not imply $C(v)$. This is crucial for ensuring that *Imply high* rule is established at the end of GROUPMOVE.

1: **function** LDDSWAPINPLACE $(A_T\ x\ ,\ A_T\ y)$
2: replace every LDD $F{:}(x, H, L)$ with (y, G_1, G_0), where
3: F_{00}, F_{01} are the $\neg y$ and y cofactors of L
4: F_{10}, F_{11} are the $\neg y$ and y cofactors of H
5: $F'_{11} \leftarrow IMP(y, C(F_{11}))\ ?\ H(F_{11}) : F_{11}$
6: $F'_{01} \leftarrow IMP(y, C(F_{01}))\ ?\ H(F_{01}) : F_{01}$
7: **if** $(F'_{11} = F'_{01} \vee (IMP(x, C(F'_{01})) \wedge F_{11} = H(F'_{01})))$ **then**
8: $G_1 = F'_{01}$
9: **else** $G_1 \leftarrow (x, F'_{11}, F'_{01})$
10: **if** $(F_{00} = F_{10} \vee (IMP(x, C(F_{00})) \wedge F_{10} = H(F_{00})))$ **then**
11: $G_0 \leftarrow F_{00}$
12: **else** $G_0 \leftarrow (x, F_{10}, F_{00})$

Fig. 5. Swapping adjacent labels in an LDD.

```
1: function BBQE (LDD f, P ⊆ T-atoms, V ⊆ Var)
2:     if (f = 1) then return TOTDD (THQELIM (V, P))
3:     if (f = 0 ∨ THUNSAT(P)) then return 0
4:     c ← C(f)
5:     if (V ∩ VARS(c) ≠ ∅) then
6:         t ← BBQE(H(f), P ∪ {c}, V)
7:         e ← BBQE(L(f), P ∪ {¬c}, V)
8:         return OR(t, e)
9:     else
10:        t ← BBQE(H(f), P, V)
11:        e ← BBQE(L(f), P, V)
12:        return ITE (c, t, e)
```

Fig. 6. Black-box QELIM.

Correctness of GROUPMOVE . Let $G = (V, E)$ be a multi-rooted DAG of LDDs. A set of T-atoms X is *closed for* G iff whenever $x \in X$ and there is a $v \in V$ s.t. $IMP(C(v), x)$ or $IMP(x, C(v))$, then $C(v) \in X$. The correctness follows from Theorem 1.

Theorem 1 *Let $G = (V, E)$ be a multi-rooted DAG of ROLDDs, and X, Y be two non-empty sets of T-atoms that are closed and adjacent in G. Let $G' = (V', E')$ be the DAG after* GROUPMOVE(X, Y). *Then, (i) G' is a DAG of ROLDDs, and (ii) for any node v in G and the same node v' in G', $\exp(v) \Leftrightarrow \exp(v')$.*

IV. EXISTENTIAL QUANTIFICATION FOR LDDS

In this section, we describe three techniques for QELIM over LDDs. In the algorithms, TOTDD is used to convert a set P of T-atoms into an LDD for $\bigwedge P$; ITE and OR mean the corresponding LDD operations. For simplicity, we do not distinguish between a single T-atom and the corresponding LDD.

A. Black-box QELIM

Our black-box QELIM (see Fig.6) is called BBQE. It applies an external theory QELIM to each path of an LDD. BBQE requires the following helper functions from the theory:

- THUNSAT(P) decides whether $\bigwedge P$ is unsatisfiable;
- THQELIM(V, P) given a set of variables V and a set of T-atoms P, computes a set of T-atoms P' s.t. $\bigwedge P'$ is equivalent to $\exists V \,.\, \bigwedge P$.

Running time of BBQE(f, P, V) is linear in the number of paths of f. BBQE is not compatible with dynamic programming since it propagates the context P along every branch of an LDD.

B. White-box QELIM

Our white-box QELIM applies Fourier-Motzkin (FM) elimination directly to LDDs. It extends the QELIM algorithm of DDDs [16] to any adequate fragment T of LA.

First, we briefly recall FM elimination [3]. Let φ be a conjunction of T-atoms, and x be a numeric variable. FM elimination of x from φ proceeds as follows: Initially S is the set of all atoms of φ, and $S' = \emptyset$. For each $p \in S$, remove p from S. If $x \notin$ VARS(p) then add p to S'; otherwise, for each $t \in S$ s.t. $x \in$ VARS(t) add RSLV(p, x, t) to S'. Note

```
1: function WBQE1 (x ∈ Var, LDD f)
2:     if (f = 1 ∨ f = 0) then return f
3:     if (x ∉ VARS(C(f))) then
4:         return ITE(f, WBQE1(x, H(f)), WBQE1(x, L(f)))
5:     t ← WBQE1(x, DR({C(f)}, x, H(f)))
6:     e ← WBQE1(x, DR({¬C(f)}, x, L(f)))
7:     return OR (t,e)
```

Fig. 8. Basic white-box QELIM.

that RSLV(p, x, t) is defined to be TRUE when x occurs in the same phase in both p and t. Upon termination, $\bigwedge S'$ is equivalent to $\exists x \,.\, \varphi$.

The key insight of white-box QELIM is to apply FM *simultaneously* to every $\mathbf{1}$-path of an LDD. The main step is simultaneous resolution. It is done by a function DR(S, x, f) (short for DAGRESOLVE) that takes a set S of T-atoms, a variable x, and an LDD f, and returns an LDD obtained by adding to each $\mathbf{1}$-path π of f the resolvents of S and π on x.

DR is implemented as follows: if f is a constant, it returns f; otherwise, it applies one of the recurrences shown in Fig. 7.

When implemented with dynamic programming, the number of recursive calls to DR is linear in the number of nodes in f. However, at the end it needs to restore orderedness, which, like in DDDs, is exponential in the size of f in the worst case.

The basic white-box algorithm WBQE1(x, f) is shown in Fig. 8. At each iteration, the algorithm either descends to branches of f (line 4), or removes a top-node of f whose label contains x (lines 5–7). The algorithm terminates since at each iteration the number of nodes labeled with an atom containing x decreases.

Recall that variable ordering in LDDs always respects chains of implications among T-atoms. Consider a "low implication chain" of LDD nodes $u_1, \ldots, u_n$. That is,

$$\forall 1 \le i < n \,.\, L(u_i) = u_{i+1} \wedge C(u_i) \Rightarrow C(u_{i+1}) \,.$$

Then, $\forall 1 \le i \le n \,.\, \neg C(u_n) \Rightarrow \neg C(u_i)$. Let c be a T-atom and $x \in Var$. Then,

$$\forall 1 \le i \le n \,.\, \text{RSLV}(\neg C(u_n), x, c) \Rightarrow \text{RSLV}(\neg C(u_i), x, c) \,.$$

Let S be a set of T-atoms. Then,

$$\text{RSLV}(\{\neg C(u_1), \ldots, \neg C(u_n)\} \cup S, x, c) \Leftrightarrow \text{RSLV}(\{\neg C(u_n)\} \cup S, x, c) \,.$$

We use this observation in the algorithm WBQE2 (see Fig. 9) to reduce the number of calls to DR.

Since WBQE2(x, f) uses DR, in the worst case it is exponential in the number of nodes in f. However, in the best case it has the same complexity as BDD QELIM.

C. Eliminating Multiple Variables

Unlike BBQE, white-box QELIM only eliminates one variable at a time – this is a fundamental limitation of Fourier-Motzkin. When eliminating multiple variables, the order in which they are eliminated is crucial. For example, consider a formula

$$x - y \le 1 \wedge z - x \le 2 \wedge w - z \le 3 \,.$$

$$\frac{\text{DR}(S, x, \text{ITE}(u, v, w)) \qquad x \notin \text{VARS}(u)}{\text{ITE}(u, \text{DR}(S, x, v), \text{DR}(S, x, w))}$$

$$\frac{\text{DR}(S, x, \text{ITE}(u, v, w)) \qquad x \in \text{VARS}(u)}{(\text{RSLV}(S, x, u) \wedge u \wedge \text{DR}(S, x, v)) \vee (\text{RSLV}(S, x, \neg u) \wedge \neg u \wedge \text{DR}(S, x, w))}$$

Fig. 7. Two recurrences used by DR.

```
1: function WBQE2 (x ∈ Var, LDD f)
2:     if (f = 1 ∨ f = 0) then return f
3:     if (x ∉ VARS(C(f))) then
4:         return ITE(f, WBQE2(x, H(f)), WBQE2(x, L(f)))
5:     S ← ∅; K ← ∅
6:     repeat
7:         c ← C(f); S ← S ∪ {c}
8:         d ← DR(S, x, H(f))
9:         K ← K ∪ {WBQE2(x, d)}
10:        S ← {¬c} ; c' ← C(L(f))
11:        if (IMP(c, c')) then f ← L(f);
12:    until (¬IMP(c, c'))
13:    d ← DR(S, x, L(f))
14:    K ← K ∪ {WBQE2(x, d)}
15:    return OR (K)
```

Fig. 9. Improved white-box QELIM.

```
1: function WBMVQE (V ⊆ Var, LDD f)
2:     res ← f
3:     while (V ≠ ∅) do
4:         V' ← FINDDROPVARS(V, res)
5:         if (V' ≠ ∅) then
6:             res ← DRVAR(V', res)
7:             V ← V \ V'
8:             continue
9:         x ← CHOOSEVAR(V, res)
10:        res ← WBQE2(x, res)
11:        V ← V \ {x}
12:    return res
```

Fig. 10. White-box QELIM for multiple variables.

Assume we want to eliminate x and y. There are two elimination orders: (a) x, y, and (b) y, x. In case (a), first $x - y \leq 1$ is removed and resolved with $z - x \leq 2$ to get: $z - y \leq 3 \wedge z - x \leq 2 \wedge w - z \leq 3$. Then, $z - x \leq 2$ is dropped, and finally $z - y \leq 3$ is dropped to get: $w - z \leq 3$. In case (b), first $x - y \leq 1$, and then $z - x \leq 2$ are dropped. No resolution is needed.

This example highlights two things. First, eliminating variables that occur in fewer atoms (like y above) leads to fewer resolutions and potentially smaller intermediate results. Second, variables that occur in a single atom (like y above), or, more generally, variables that occur in pure polarity in a disjunct of a DNF, are eliminated by dropping their atoms without any resolution steps. Since there is no resolution, multiple such variables can be eliminated at once.

We use these observations in a multi-variable elimination strategy called WBMVQE shown in Fig. 10. WBMVQE is parametrized by two functions:

- FINDDROPVARS(V, f) returns the subset of variables in V that do not need to be resolved on when they are eliminated from an LDD f.
- CHOOSEVAR(V, f) chooses a variable in V to be eliminated from an LDD f.

```
1: function DRVAR (V ⊆ Var, LDD f)
2:     if (f = 1 ∨ f = 0) then return f
3:     if (VARS(C(f)) ∩ V = ∅) then
4:         return ITE(C(f), DRVAR(V, H(f)), DRVAR(V, L(f)))
5:     else
6:         return OR(DRVAR(V, H(f)), DRVAR(V, L(f)))
```

Fig. 11. Dropping variables from an LDD without resolution.

It also uses a helper function DRVAR(V, f) shown in Fig. 11 that disjunctively drops all nodes from an LDD f that are labeled with an atom containing a variable in V.

WBMVQE iterates through two phases. First, it eliminates all variables that do not need resolution (lines 4–8). This is repeated until no more variables can be dropped. Second, it eliminates a variable using WBQE2 (lines 9–10). This process repeats until all variables are eliminated.

Note that WBMVQE is only a strategy – it must be instantiated with an implementation of FINDDROPVARS and CHOOSEVAR. For our experiments, we have instantiated WBMVQE by counting the number of occurrences of variables in V in the atoms labeling the nodes of f (i.e., the support of f). In our case, FINDDROPVARS returns all variables that occur in only a single atom; CHOOSEVAR picks a variable randomly from all variables that occur in the least number of atoms.

V. EXPERIMENTAL RESULTS

We have implemented LDDs within CUDD. Except for DVO, our implementation is an external library that adds, but does not modify, CUDD. Following CUDD, we use *complemented edges* which allow for constant time negation. For DVO, we reuse all CUDD's heuristics, but modify how adjacent labels are swapped (see Sec. III). Fortunately, CUDD already can group variables and sift groups simultaneously (i.e., GROUPMOVE), but we had to change its implementation of GROUPMOVE to sift top-down instead of bottom-up.

To evaluate our implementation, we used a benchmark derived from 25 real C programs, including `mplayer`, `ff_mpeg`, `gzip`, `tcsh`, and `CUDD`. We created the benchmark by compiling each program using LLVM [14] with optimization, loop unrolling, and inlining enabled, and then approximating the SSA control-flow graph of each function by a UTVPI formula. In general, our approximation is unsound since we replaced LA formulas by UTVPI. However, each such formula is similar in structure to a bounded model-checking problem, e.g., as in CBMC [7]. We narrowed down the initial 10,000 formulas to our benchmark of 850 using the criteria: (a) more than 1,000 nodes in LDD or BDD representation, or (b) more than 2 seconds to build an LDD or a BDD, or (c) more than 2 seconds to solve with SVO.

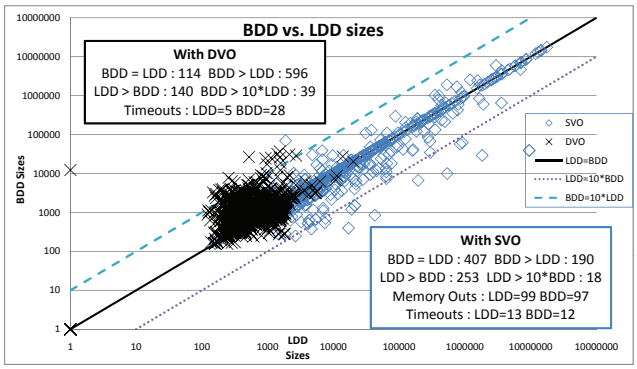


Fig. 12. Size comparison in DD nodes between LDD and BDD.

These formulas range in sizes from 4KB to 700KB, and have between 30 to 7,956 variables.

We conducted two sets of experiments to evaluate the effect of local LDD reductions and the effectiveness of QELIM, respectively. All experiments were done on an 8-core 2.00GHz Xeon with 3GB of RAM. Each test-case was limited to 300s of CPU and 512MB of RAM. The benchmark and detailed experimental results are available at `www.sei.cmu.edu/staff/chaki/FMCAD-09.html`.

Local Reductions. To measure the effect of local reductions, we compared the sizes of LDD v.s. BDD for each test-case using both SVO and DVO. By "BDD" we mean a BDD constructed by abstracting all UTVPI predicates by propositional variables. For BDDs, SVO is a syntactic variable order where each new predicate is placed at the end of the current order. It is the same for LDDs, except when it violates the LDD ordering restriction: in that case, the new predicate is inserted at an appropriate position in the order. For example, for the formula $x - y \leq 5 \wedge y - z \leq 0 \wedge x - y \leq 15$, the BDD SVO is as seen in the formula, while the LDD SVO is $x - y \leq 5, x - y \leq 15, y - z \leq 0$. DVO means that automatic DVO was enabled and DD manager reordered whenever memory usage was high.

The results are summarized in the scatter plot in Fig. 12. It shows LDD sizes (x-axis) vs. BDD sizes (y-axis). For example, the point $(325, 720)$ means that some test-case had an LDD with 325 nodes and a BDD with 720 nodes. Both axes are in log-scale. SVO and DVO experiments are marked by diamonds and crosses, respectively. As expected, DVO leads to significantly smaller diagrams for both BDDs and LDDs. Interestingly, with DVO LDDs are significantly smaller (sometimes by an order of magnitude) than BDDs, whereas with SVO the situation is reversed. This validates our intuition that DVO is even more significant for LDDs than for BDDs.

Quantification. To evaluate the effectiveness of our QELIM algorithms, we measured the time to quantify out the first $4/5^{th}$ syntactically appearing variables in each test-case. This roughly corresponds to a forward-image (a.k.a. strongest post-condition) in program analysis. We compared BBQE and 4 variants of WBMVQE (DVO is used unless stated otherwise):

TABLE I
Overall results for QELIM algorithms. All times are in seconds. *Total* = total times; *QE* = QELIM time; *TO* = Timeout; *MO* = Memoryout. No MOs for the Easy cases. No *Total*/*QE* for BB+DVO since it TO in most cases.

	Hard (154 cases)				Easy (696 cases)		
Alg.	*Total*	*QE*	*TO*	*MO*	*Total*	*QE*	*TO*
BBQE	—	—	141	0	—	—	670
WB+DVO	10,953	3,329	9	0	784	219	0
WB+SVO	38,739	36,511	21	99	395	80	0
WBWB1	11,047	3,761	11	0	829	264	0
WB-DV	17,043	13,358	34	0	5,649	5,151	8

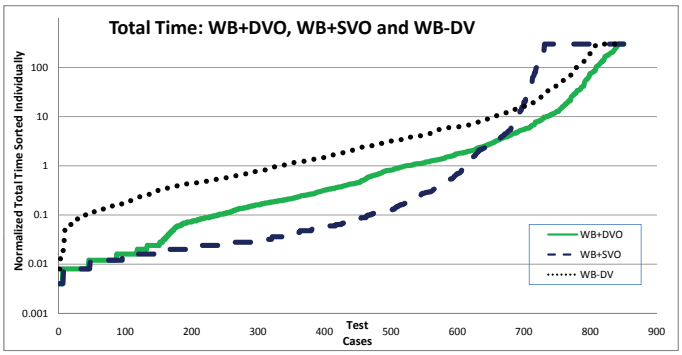


Fig. 13. Total times for white-box QELIM.

- WB+SVO: with SVO,
- WB+DVO: with DVO,
- WB-DV: with lines 4–8 removed, and
- WBWB1: with a call to WBQE2 on line 10 replaced with the call to WBQE1.

The overall results are summarized in Table I. We call the 696 test-cases that are solved by WB+SVO in under 15s *easy*, and the remaining 154 *hard*. *Total* is the time including parsing, building an LDD, and QELIM; *QE* is the time spent in BBQE or WBMVQE. All times are in seconds. Each failure is normalized to 300s. Size of the final LDDs ranged from $140,092$ to 1.

BBQE performed the worst, solving only 38 cases. This is not surprising. The average path-size in the benchmark is 1.4×10^{131}, and, in 9 cases, it cannot even be represented with a C `double`. Furthermore, enumerating all paths of an LDD succeeds only in 155 cases.

WB+DVO and WB+SVO performed the best for hard and easy cases, respectively. Thus, for hard cases, the extra time spent searching for a better order pays off.

The chart in Fig. 13 shows the total time (y-axis, in log-scale) for white-box QELIM, sorted individually in increasing order. For harder cases, the time increases gradually with DVO (solid and dotted series, and WBWB1, which is not shown, but is similar to WB+DVO). However, the time jumps dramatically for SVO (dashed series). This indicates that DVO is more robust than SVO during QELIM. This is further illustrated by the chart in Fig. 14 that compares WBMVQE under SVO and DVO. In the chart, y-axis is the total time in log-scale, and the x-axis of both series is sorted by WB+SVO time. For

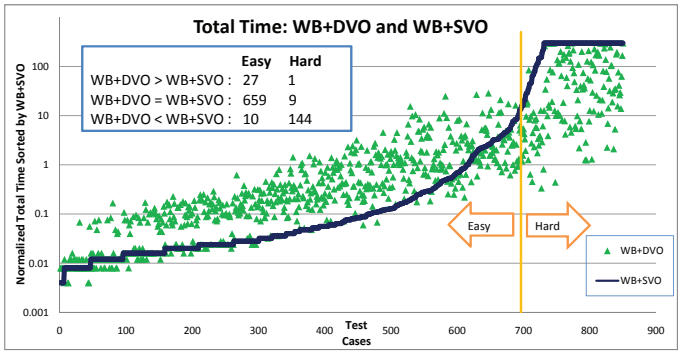


Fig. 14. Total time comparison between WB+SVO and WB+DVO.

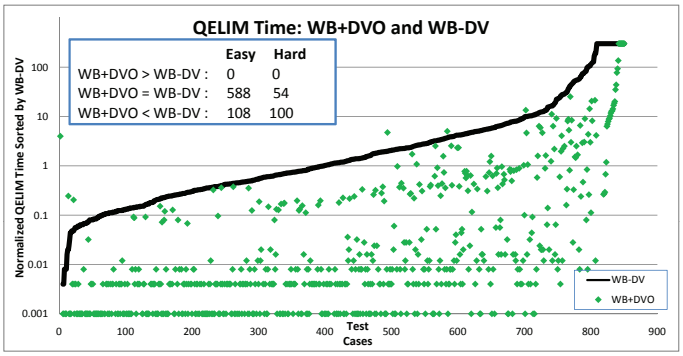


Fig. 15. QELIM time comparison between WB-DV and WB+DVO.

hard instances, in many cases WB+DVO is several orders of magnitude faster than WB+SVO. Note that due to log-scale this difference is much more significant than it appears.

The chart in Fig. 15 compares the effect of multiple-variable elimination heuristic (lines 4–8 of WBMVQE). In the chart, y-axis is the QELIM time in log-scale, and the x-axis of both series is sorted by WB-DV time. From the chart, WB+DVO almost always outperforms WB-DV, often by several orders of magnitude. Overall, WB+DVO is about 5 times faster than WB-DV for QELIM.

We also compared WBQE1 and WBQE2 (i.e., WB+DVO and WBWB1). WBWB1 timed out in 2 more cases than WB+DVO. Other than that, the two algorithms performed similarly.

We are not aware of other tools for existential quantification of arbitrary LA formulas. Thus, we have only compared between our own implementations. LDDs can be used as an SMT-solver for LA: to decide whether a formula is satisfiable first build an LDD for the formula and then quantify out all numeric variables. In our preliminary experiments this approach was not competitive with current DPLL-style SMT-solvers. We have not pursued it further.

VI. Conclusion

In this paper, we have tackled the problem of space-efficient representation of LA formulas with support for Boolean operations and QELIM. This problem is at the heart of many program analysis tasks. To this end, we have extended Difference Decision Diagrams to Linear Arithmetic. Our key contributions are: support for Dynamic Variable Ordering, QELIM algorithms, an implementation inside CUDD, and empirical evaluation on a large benchmark derived from real programs.

Overall, we found that LDDs in combination with DVO and dynamic programming-based QELIM algorithm leads to an effective data structure for program analysis tasks.

We believe that LDDs is a good basis for a combined predicate and numeric abstract domain, in the style of [11]. We plan to explore this direction in future work.

References

[1] H. R. Andersen and H. Hulgaard. Boolean Expression Diagrams, 1997.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. *FMSD*, 10(2/3):171–206, 1997.

[3] A. Bik and H. Wijshoff. Implementation of Fourier-Motzkin Elimination. Technical Report 94-42, Dept. of Computer Sci., Leiden University, 1994.

[4] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[5] R. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *DAC'95*, June 1995.

[6] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD'07*, pages 69–76, 2007.

[7] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS'04*, volume 2988 of *LNCS*, pages 168–176, 2004.

[8] E. Felt, G. York, R. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering for BDD Minimization. In *EURO-DAC'93/EURO-VHDL'93*, pages 130–135, 1993.

[9] J. F. Groote and O. Tveretina. Binary Decision Diagrams For First-order Predicate Logic. *J. Log. Algebr. Program.*, 57(1-2):1–22, 2003.

[10] J. F. Groote and J. van de Pol. Equational Binary Decision Diagrams. In *LPAR'00*, pages 161–178, 2000.

[11] A. Gurfinkel and S. Chaki. Combining Predicate and Numeric Abstraction for Software Model Checking. In *FMCAD'08*, 2008.

[12] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV'06*, volume 4144 of *LNCS*, pages 413–426, 2006.

[13] K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock Difference Diagrams. *Nord. J. Comput.*, 6(3):271–298, 1999.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*, March 2004.

[15] A. Miné. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006.

[16] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *Proceedings of 13th International Conference on Computer Science Logic*, volume 1683 of *LNCS*, pages 111–125, 1999.

[17] S. Panda, F. Somenzi, and B. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *ICCAD'94*, pages 628–631, 1994.

[18] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD'93*, pages 42–47. IEEE, 1993.

[19] K. Strehl and L. Thiele. Symbolic Model Checking of Process Networks Using Interval Diagram Techniques. In *ICCAD'98*, pages 686–692, 1998.

Efficient Decision Procedure for Non-linear Arithmetic Constraints using CORDIC

Malay K. Ganai Franjo Ivančić
NEC Laboratories America, Princeton, NJ, USA

***Abstract*—In verification of hybrid discrete-continuous and embedded control systems, one encounters decision problems involving non-linear constraints. We propose an efficient decision procedure (CORD) for such decisions problems using CORDIC algorithms, and an off-the-shelf SMT(LA) (Satisfiability Modulo Theory for Linear Arithmetic) solver, for given precision requirements. We first translate the non-linear part of the decision problem to a SMT(LA) formula using CORDIC algorithms, accounting for all the inaccuracies safely. In the translation, we use a normalization scheme, combined with interval bounds to obtain a linearized formula without compromising the precision requirements. On such a linearized formula, we devise a DPLL-style Interval Search Engine (DISE) that explores various combinations of interval bounds using a SMT(LA) solver. In our experiments, we demonstrate the efficacy of our approach, and compare it with a latest state-of-the-art decision procedure.**

I. Introduction

Verification of hybrid discrete-continuous and embedded control systems requires solving decision problems comprising Boolean combination of linear and non-linear operations, involving transcendental and algebraic functions. In general, such a decision problem is undecidable. Therefore, one can hope that a solver can provide soundness guarantee, wherein soundness implies that the solver does not err on the unsatisfiability decision. It can also provide completeness guarantee up to a given non-zero error bound. In other words, the solver can err on the satisfiability decision, provided the spuriousness is due to numerical inaccuracy with in the given error bound. Such guarantees are useful for analyzing numerical stability of such systems implemented using imprecise arithmetic.

There are many advanced tools in the field of OR (Operation Research) for solving linear and non-linear arithmetic optimization and feasibility problems. However, even for linear problems these tools can give incorrect results due to the internal use of floating-point arithmetic (which is imprecise due to unavoidable rounding errors). In general, for OR applications such inaccuracies are acceptable. However, for verification applications, exactness in the results is important, especially, in dealing with strict inequalities and dis-equalities.

With a growing usage of high-level design abstraction to capture today's complex design features, the focus of verification techniques such as Bounded Model Checking (BMC) [1], [2] has been shifting from the propositional reasoning to decision procedures known as Satisfiability Modulo Theory (SMT) solvers [3]–[5]. A SMT problem for a decidable first-order equation theory $\mathcal{T}$, denoted as $SMT(\mathcal{T})$, comprises Boolean combination of theory $\mathcal{T}$; given a formula ϕ, determine whether ϕ is $\mathcal{T}$-satisfiable, i.e., whether there exists a model of $\mathcal{T}$ that is also a model of ϕ. In general, SMT-solvers use precise arithmetic, without compromising the accuracy. Previous SMT-style decision procedures for non-linear problems [6], [7], however, give imprecise results due to the use of floating-point arithmetic in the solvers.

A. Our Approach: Overview

We propose an efficient decision procedure $\mathcal{CORD}$ for non-linear arithmetic constraints, leveraging the advancement in SMT solvers for the theory of linear arithmetic ($\mathcal{LA}$). We first translate the non-linear part of such a decision problem to a SMT($\mathcal{LA}$) formula using CORDIC algorithms [8], [9], accounting for all inaccuracies safely. Further, we use a normalization scheme, combined with interval bounds to obtain a linearized formula without compromising the precision requirements. On such a linearized formula, we devise a DPLL-style [10] Interval Search Engine ($\mathcal{DISE}$) that explores various combinations of interval bounds using an off-the-shelf state-of-the-art SMT($\mathcal{LA}$)-solver that uses precise arithmetic. We present an overview of our decision procedure $\mathcal{CORD}$ as shown in Figure 1. Our main contributions are:

- Our approach uses an off-the-shelf SMT($\mathcal{LA}$)-solver, thereby, benefits directly from the ongoing advancements of the solvers [11], unlike an approach [7].
- We present novel strategies for guiding interval bounds in a DPLL-style interval search. We introduce the notion of *hard* and *soft* constraints to guide the choice of interval bounds using *theory learning* and *theory suggestion* in an incremental formulation.
- Our approach is both sound, and complete for given non-zero error bounds, and finite interval bounds.

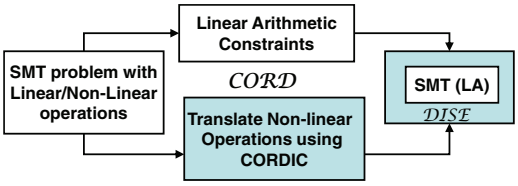


Fig. 1. $\mathcal{CORD}$: Decision procedure for real linear and non-linear operations

B. Related Work

We discuss solvers for non-linear decision problems proposed previously and their limitations.

`ABSolver`. In the tool ABsolver [6], a central controller is integrated with theory-specific off-the-shelf solvers for each of the following theory: Boolean, linear, and non-linear. The controller coordinates the entire solving process and delegates the currently active constraints to the respective theory solvers. For non-linear constraints, it uses a numerical optimization tool IPOPT [12], [13]. Since IPOPT is inherently imprecise (i.e., uses floating point arithmetic), it is likely that the results are inexact, neither sound nor complete.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

iSAT. In another approach iSAT [7]—implemented in a DPLL-style framework—the conjunction of linear/non-linear constraints are solved using interval constraint propagation (ICP) engine. For interval arithmetic, iSAT uses a floating-point arithmetic. Clearly, such an approach gives spurious result even without non-linear constraints. For instance, the latest version (*ver-0.8.5*) of iSAT [14] returns the following conjunction of linear constraints satisfiable:

$$(x+y<a)\wedge(x-y<b)\wedge(2\cdot x>a+b)\wedge(a=1)\wedge(b=0.1) \quad (1)$$

However, the constraints are unsatisfiable, as the first two and the last two constraints imply $(2x < 1.1)$; which is clearly inconsistent with the third constraint. The authors of iSAT claim that the approach is sound, which is reportedly achieved by rounding the floating point results *outwards* during ICP. However, for the following satisfiable conjunction of linear constraints, we found that iSAT gave unsound results, i.e., it reported the conjunction unsatisfiable.

$$(x \leq 10^9) \wedge (x + p > 10^9) \wedge (p = 10^{-8}) \quad (2)$$

We believe that such unsound results are not acceptable for verification applications. Interestingly, when we replaced the last constraint $p = 10^{-8}$ by $p = 10^{-7}$, the result was reported satisfiable correctly. We believe that obtaining a sound implementation using floating-point arithmetic has many challenges. In contrast, we overcome the above anomalies using an off-the-shelf SMT($\mathcal{LA}$)-solver that uses precise arithmetic.

LBR. In a closely related approach [15], the authors have combined translation of non-linear integer decision problem into a SMT($\mathcal{LIA}$)-formula, with an iterative lazy bounding refinement (LBR) algorithm built over SMT($\mathcal{LIA}$)-solver. However, such an approach is limited to bounded integer arithmetic, and therefore, cannot be applied directly to handle non-linearity over reals.

Outline In Section II, we give relevant background information, followed by discussion of $\mathcal{CORD}$ in Sections III-VI. Specifically, we present a translation of non-linear to linear terms in III, encoding of such a translation to SMT($\mathcal{LA}$) in IV, discussion on numerical accuracy of such an encoding in V, and a search procedure $\mathcal{DISE}$ in VI. We present our experimental results in Section VII, and conclusions in Section VIII.

II. Background

A. Mixed Expressions

We consider decision problems with Boolean expressions *bool-expr*, integer term expressions *int-expr* and real term expressions *real-expr*. We use *term-expr* to denote either *int-expr*, or *real-expr*. We use "$\cdot$" for constant multiplication, and $\times$ for non-linear multiplication.

Real term expressions comprise—linear operations such as $+, -, \cdot$; non-linear algebraic operations such as $\times, /$; non-linear transcendental operations such as $\sin, \cos$, etc.— on two real constants $\mathbb{R}$, *real-var* (real variables) or *real-expr*.

Integer term expressions includes linear operations such as $+, -, \cdot$ on two integer constants $\mathbb{Z}$, *int-var* (integer variables) or *int-expr*.[1]

[1]Here, we limit our discussion to integer linear operations. However, we can combine our presented approach with [15] to handle non-linear integer operations as well.

Boolean expressions include Boolean operations such as $\wedge, \vee, \neg$ on two Boolean constants $\{false, true\}$, *bool-var* (Boolean variables) or *bool-expr*; relation operations $\bowtie \in \{=, \leq, <, \geq, >\}$ on two *term-expr* terms.

We use ITE(*bool-expr*, *term-expr*, *term-expr*) as a shorthand to denote the "if-then-else" operator. We use $\mathcal{B}$, $\mathcal{Z}$, and $\mathcal{R}$ to denote the set of *bool-expr*, *int-expr* and *real-expr*, respectively, in a given decision problem. These expressions are interpreted over some valuations $\sigma = \sigma_B \times \sigma_Z \times \sigma_R$, where $\sigma_B : \mathcal{B} \rightarrow \{false, true\}$, $\sigma_Z : \mathcal{Z} \rightarrow \mathbb{Z}$, and $\sigma_R : \mathcal{R} \rightarrow \mathbb{R}$. We use $\sigma(x)$ to denote the value of an expression $x \in \mathcal{B} \cup \mathcal{Z} \cup \mathcal{R}$. We use $\phi \Longrightarrow_{SAT} \psi$ to denote that satisfiablility of ϕ implies that of ψ, and $\phi \Longleftrightarrow_{SAT} \psi$ to denote their equisatisfiability.

B. $\mathcal{LA}$-theory and SMT($\mathcal{LA}$)

A theory of $\mathcal{LA}$ constitutes a conjunction of linear arithmetic constraints (LAC) $\sum_i a_i \cdot x_i \leq d$, where a_i is a rational constant. There has been extensive research on checking the decidability of LAC, in both operational research and verification applications. However, there is a subtle difference in the goals of the $\mathcal{LA}$ solvers in the respective domains. The truthfulness of the outcome is not so critical in the former application, but it is a necessary requirement in the latter. To meet the requirement, the solvers for latter approaches typically use rational arithmetic (a precise arithmetic), which usually incurs high computation cost. To overcome that, researches have attempted to minimize rational arithmetic operations with floating-point arithmetic with error correction [16].

A SMT($\mathcal{LA}$) problem is a problem of deciding satisfiability of Boolean expressions obtained on applying Boolean connectives on propositional atoms and relational operators on two $\mathcal{LA}$-terms, wherein a $\mathcal{LA} - term$ is a *int-expr* or *real-expr* build from linear operators $(+, -, \cdot)$. For our decision procedure, we only require that a SMT($\mathcal{LA}$)-solver decides *exactly* on a given SMT($\mathcal{LA}$) formula. Specifically, we focus on DPLL-based SMT($\mathcal{LA}$)-solvers [3], [4] that has incremental solving capabilities. These solvers typically provide interfaces to incrementally assert/retract inequalities of the form $\underline{x_i} \leq x_i \leq \overline{x_i}$, (also denoted as $x_i \in [\underline{x_i}, \overline{x_i}]$).

C. CORDIC algorithms

CORDIC (**CO**ordinate **R**otation **D**igital **C**omputer) algorithms were first proposed by Volder in 1956 [8] and further extended by Walther [9] for computing many elementary functions, using only adders and shifters in finitely many iterative steps. Such an efficient computation has been widely used in several signal processing computing algorithms such as Fourier transformation, and lattice filtering. The basic task performed is to rotate a vector in a 2D-plane through an angle in a linear, circular or hyperbolic coordination system.

There are essential two modes of operations, *vectoring* (a.k.a backward), and *rotation* (a.k.a forward). In vectoring mode, a given vector (x, y) (represented in a 2D-plane) is rotated to X-axis to obtain a new coordinate $(x', y') = (\sqrt{x^2 + y^2}, 0)$, so that the length of the vector, and angle of inclination can be computed. In the rotation mode, the vector (x, y) is rotated over a given angle α to compute a new coordinate (x', y').

The key point in such computation is to approximate a rotation with a sequence of micro-rotations α_k with $k \in \{0, \cdots, n-1\}$, $0 < \alpha_k \leq \frac{\pi}{2}$ such that the new coordinate

can be computed easily using adders and shifters. Using such a simple principle, one can compute a wide range of elementary functions *approximately* using a finite linear structure.

A unified algorithm [9] providing the basic computation using triplets (x_k, y_k, z_k) with $0 \leq k < n$ is described in the following. These triples define the general recursion for computing elementary functions, wherein (x_0, y_0) is needed to be rotated by an angle z_0 in rotation mode, and to X-axis in vectoring mode.

$$x_{k+1} = x_k - c \cdot \delta_{c,k} \cdot y_k \cdot 2^{-\tau_{c,k}} \quad (3)$$

$$y_{k+1} = y_k + \delta_{c,k} \cdot x_k \cdot 2^{-\tau_{c,k}} \quad (4)$$

$$z_{k+1} = z_k - \delta_{c,k} \cdot \alpha_{c,k} \quad (5)$$

In equations above, $c \in \{0, +1, -1\}$ with $c = 0$ corresponding to a linear rotation, i.e., to obtain product or quotient; $c = 1$ corresponding to a circular rotation, i.e., to obtain $\sin$, $\cos$, etc.; $c = -1$ corresponding to a hyperbolic rotation, i.e., to obtain $\sinh$, $\cosh$, $\exp$, $\log$, etc. $\tau_{c,k}$ is a non-decreasing integer shift sequence satisfying $\tau_{c,k} \leq \tau_{c,k+1} \leq \tau_{c,k} + 1$, that is chosen uniquely for a given c. The angle rotation in the k^{th} iteration is

$$\alpha_{c,k} = c^{-1/2} tan^{-1}(c^{1/2} \cdot 2^{-\tau_{c,k}}) \quad (6)$$

$\delta_k (= \pm 1)$ are chosen during the iteration so that the desired value is reached. In vectoring mode, the sign of δ_k is the same as $x_k \times y_k$, such that $|y_{k+1}| < |y_k|$; while for rotating mode, the sign of δ_k is the same as z_k, such that $|z_{k+1}| < |z_k|$.

Consider the vectoring mode. The choice of δ_k ensures that

$$|z_{k+1}| = ||z_k| - \alpha_{c,k}| \quad (7)$$

For z to converge within $\alpha_{c,n-1}$, following two conditions should be satisfied [8], [9].

Condition 1: The total rotation angle can not exceed the total sum of rotations, denoted as α_{max}.

$$|z_0 - z_n| \leq \alpha_{max} \stackrel{def}{=} \sum_{k=0}^{n-1} \alpha_{c,k} \quad (8)$$

This constrains the values which x_0, y_0 may take, limiting the *domain of convergence*.

Condition 2: The sum of remaining rotations at each step be sufficient to bring the angle to at least within $\alpha_{c,n-1}$, i.e.,

$$\alpha_{c,k} - \sum_{j=k+1}^{n-1} \alpha_{c,j} < \alpha_{c,n-1} \quad (9)$$

Similar conditions apply for the rotation mode [8], [9].

III. Translation of Non-linear to Linear terms

In this paper, we focus only on translation of non-linear operations involving multiplication and division, obtained using linear rotations of CORDIC algorithms.

A. Multiplication

A multiplication $p = s \times t$ can be encoded into a Boolean combination of LAC using (3-5) in rotation mode with $c = 0$, $(x_0 = s, y_0 = 0, z_0 = t)$, $\alpha_{0,k} = 2^{-k}$ (as $\lim_{c \to 0} \alpha_{c,k} = 2^{-\tau_{0,k}}$) and $\tau_{0,k} = k$ as follows:

$$y_{k+1} = y_k + \delta_{0,k} \cdot x_k \cdot 2^{-k} \quad (10)$$

$$z_{k+1} = z_k - \delta_{0,k} \cdot 2^{-k} \quad (11)$$

where $\delta_{0,k} = ITE(z_k \geq 0, 1, -1)$. Note, $x_{k+1} = x_k$. The n^{th} iteration value y_n gives the "CORDIC approximation" to $s \times t$, denoted as $cordicMult(s, t)$. The domain of convergence criteria (8) is $t \in (-2, 2)$. (Note, $\alpha_{max} = \Sigma_{k=0}^{n-1} 2^{-k} < 2$ for any $n > 0$) For this input domain, an absolute error (approximation error [17]) incurred is:

$$err_{abs} \stackrel{def}{=} |y_n - p| = |z_n| \times |x_0| \leq 2^{-(n-1)} \cdot |x_0| \quad (12)$$

where p is the exact result. The upper bound on err_{abs} is referred to as *absolute error tolerance*[2]. A relative error in the computation (10-11) is given as follows:

$$err_{rel} \stackrel{def}{=} \frac{|y_n - p|}{|p|} = \frac{|z_n|}{|t|} \leq \frac{2^{-(n-1)}}{|t|} \quad (13)$$

The upper bound on err_{rel} is referred to as *relative error tolerance*. Clearly, err_{rel} increases as $|t| \longrightarrow 0$. Though increasing n will reduce the err_{rel} introduced, we use a normalization scheme that bounds err_{rel}.

Example: Consider $p = s \times t$, with $s = 10$, $t = 1.575$. With $n = 8$, we have (y_0=0, z_0=1.575; y_1=10, z_1=0.575; y_2=15, z_2=0.075; $\cdots$; y_8=15.703135, z_8=0.004687), with $err_{abs} = 0.04685 \leq 2^{-8} \cdot 10 = 0.078$ and $err_{rel} = 0.003 < 2^{-8} = 0.0078$.

Normalization Scheme To bound err_{rel} for $|t| < 1$, we normalize the input $t = M_t \cdot 2^{E_t}$ such that $1 \leq |M_t| < 2$. We translate $p = s \times t$ to $cordicMult(s, M_t) \cdot 2^{E_t}$. Note, such normalization does not affect the absolute error tolerance, however, reduces the relative error tolerance. If relative error tolerance required is $2^{-(m-1)}$ where $m >> n$, we normalize the input t such that $E_t \geq -(m - n)$. Note that such a normalization meets the relative error tolerance with a small n i.e., a smaller iterative structure, which in turn, also implies a smaller translated SMT($\mathcal{LA}$) formula.

Error Correction We add the following error correction term ec_{mult} to the result y_n.

$$y'_n = y_n + \underbrace{e \times (x_0 \cdot 2^{-(n-1)})}_{ec_{mult}} \quad (14)$$

where e is a *real-var* in the interval range $[-1, 1]$, introduced for each multiplication operation.

We obtain corresponding error tolerances, as follows:

$$err'_{abs} \stackrel{def}{=} |y'_n - p| \leq 2^{-(n-2)} \cdot |x_0| \quad (15)$$

$$err'_{rel} \stackrel{def}{=} \frac{|y'_n - p|}{|p|} \leq \frac{2^{-(n-2)}}{|t|} \quad (16)$$

To maintain the precision of n, we need one additional iterative step. We later show that such an error correction term is needed to guarantee soundness to our decision procedure.

[2]As we will use exact linear arithmetic, we only need to consider approximation error, and not the rounding errors [17] in such CORDIC steps.

B. Division

A division $q = s/t$ ($t \neq 0$) can be encoded into a Boolean combination of LAC using (3-5) in *vector mode* with $c = 0$, $(x_0 = t, y_0 = s, z_0 = 0)$, $\alpha_{0,k} = 2^{-k}$ and $\tau_{0,k} = k$.

$$y_{k+1} = y_k + \delta_{0,k} \cdot x_k \cdot 2^{-k} \quad (17)$$

$$z_{k+1} = z_k - \delta_{0,k} \cdot 2^{-k} \quad (18)$$

where $\delta_{0,k} = ITE(y_k \geq 0 \otimes x_k \geq 0, -1, 1)$. ($\otimes$ is a "xnor" operation). Note, $x_{k+1} = x_k$. The resultant value z_n gives the "CORDIC approximation" to s/t, denoted as $cordicDiv(s, t)$.

The domain of convergence (8) is $(s/t) \in (-2, 2)$. For this input domain, an absolute error (*a.k.a.* approximation error [17]) introduced is:

$$err_{abs} \stackrel{def}{=} |z_n - q| = \frac{|y_n|}{|x_0|} \leq 2^{-(n-1)} \quad (19)$$

where q is the exact result. The upper bound on err_{abs} will be referred to as *absolute error tolerance*. A relative actual error in computation (17-18) is given as follows:

$$err_{rel} \stackrel{def}{=} \frac{|z_n - q|}{|q|} = \frac{|y_n|}{|y_0|} \leq \frac{2^{-(n-1)}}{|q|} \quad (20)$$

The upper bound on err_{rel} is referred to as *relative error tolerance*. Clearly, err_{rel} tends to go up as $q \longrightarrow 0$. Though increasing n will reduce the err_{rel} introduced, we use a normalization scheme that bounds err_{rel}.

Normalization To bound err_{rel} for $|q| \leq 0.5$, we normalize the inputs $s = M_s \cdot 2^{E_s}$ and $t = M_t \cdot 2^{E_t}$ such that $1 \leq |M_s| < 2$ and $1 \leq |M_t| < 2$. We translate $q = s/t$ as $cordicDiv(M_s, M_t) \cdot 2^{E_s - E_t}$. Note, such normalization does not affect the absolute error tolerance, however, reduces the relative error tolerance. If the relative error tolerance required is $2^{-(m-2)}$ where $m >> n$, we normalize the inputs s, t such that $E_s - E_t \geq -(m - n)$. Such normalization meets the relative error tolerance with a small n, and hence requiring smaller iterative structure.

Error Correction We add following error correction term ec_{div} to the result z_n.

$$z'_n = z_n + \underbrace{e \cdot 2^{-(n-1)}}_{ec_{div}} \quad (21)$$

where e is a *real-var* in the interval range $[-1, 1]$, introduced for each division instance.

We obtain corresponding error tolerances as follows:

$$err'_{abs} \stackrel{def}{=} |z'_n - q| \leq 2^{-(n-2)} \quad (22)$$

$$err'_{rel} \stackrel{def}{=} \frac{|z'_n - q|}{|q|} \leq \frac{2^{-(n-2)}}{|q|} \quad (23)$$

To maintain the precision of n, we need one additional iterative step. We later show that such an error correction is needed to guarantee soundness to the decision procedure.

IV. Encoding to SMT($\mathcal{LA}$)

We discuss the encoding of the terms introduced in the translation of non-linear to linear expressions. The encoding of the equations (10-11,17-18) to SMT($\mathcal{LA}$) is straightforward, and thus, not discussed here.

A. Encoding: Error Correction Terms ec_{mult}, ec_{div}

The term $ec_{mult} = e \times x_0 \cdot 2^{-(n-1)}$ (14) with $|e| \leq 1$, is encoded as follows:

$$(|ec_{mult}| \leq |x_0 \cdot 2^{-(n-1)}|) \quad (24)$$

Similarly, the term $ec_{div} = e \cdot 2^{-(n-1)}$ (21) with $|e| \leq 1$ is encoded as follows:

$$(|ec_{div}| \leq 2^{-(n-1)}) \quad (25)$$

Note, a constraint such as $|g| \leq |h|$ with $g, h \in \mathcal{R}$ is encoded into SMT($\mathcal{LA}$) as $(-h \leq g \leq h) \vee (h \leq g \leq -h)$.

B. Encoding: Normalization Constraints

Consider the multiplication $p = s \times t$, with $t = M_t \cdot 2^{E_t} \in [2^{E_t}, 2^{E_t+1})$ and an *int-var* E_t. Let $M_p = cordicMult(s, M_t)$ denote the result of the *normalized* CORDIC multiplication where $1 \leq |M_t| < 2$. For the given interval bound on t, we obtain the normalization constraints NCp as follows:

$$NC_p \stackrel{def}{=} (t = M_t \cdot 2^{E_t}) \wedge (p = M_p \cdot 2^{E_t}) \quad (26)$$

If $E_t < -(m - n)$, we use the interval $[-2^{-(m-n)}, 2^{-(m-n)}]$, where $t = M_t \cdot 2^{-(m-n)}$.

$$NC_p \stackrel{def}{=} (t = M_t \cdot 2^{-(m-n)}) \wedge (p = M_p \cdot 2^{-(m-n)}) \quad (27)$$

Note, $|M_t| < 1$ satisfies the domain of convergence bound.

Similarly, consider the division $q = s/t$, with $t = M_t \cdot 2^{E_t} \in [2^{E_t}, 2^{E_t+1})$, and $s = M_s \cdot 2^{E_s} \in [2^{E_s}, 2^{E_s+1})$, and *int-var*s E_t and E_s. Let $M_q = cordicDiv(M_s, M_t)$ denote the *normalized* CORDIC division where $1 \leq |M_s| < 2$ and $1 \leq |M_t| < 2$. For the given intervals for s, t we obtain the normalization constraints NC_{qn} and NC_{qd} as follows:

$$NC_{qn} \stackrel{def}{=} (s = M_s \cdot 2^{E_s}) \wedge (q = M_q \cdot 2^{E_s - E_t}) \quad (28)$$

$$NC_{qd} \stackrel{def}{=} (t = M_t \cdot 2^{E_t}) \wedge (q = M_q \cdot 2^{E_s - E_t}) \quad (29)$$

If $E_s - E_t < -(m - n)$, we use the interval $[-2^{-(m-n)}, 2^{-(m-n)}]$, where $s = M_s \cdot 2^{-(m-n)}$.

$$NC_{qn} \stackrel{def}{=} (s = M_s \cdot 2^{-(m-n)}) \wedge (q = M_q \cdot 2^{-(m-n)-E_t}) \quad (30)$$

Note, $|M_s| < 1$ and $|M_q| < 1$ satisfies the domain of convergence bound. As divide-by-zero is not well-defined, we keep t always normalized, i.e., $1 \leq |M_t| < 2$. The variables E_s and E_t define the interval bounds, and are used for interval bound search as discussed later.

C. Encoding ϕ to $\widehat{\phi}$

Consider a decision problem ϕ that contains linear terms and non-linear terms involving multiplications and divisions. Recall, $\mathcal{B}$, $\mathcal{Z}$, and $\mathcal{R}$ denote a set of all *bool-expr*, *int-expr*, and *real-expr*, respectively in ϕ. Let $\mathcal{R} = \mathcal{R}_L \cup \mathcal{R}_{NL}$, where $\mathcal{R}_L$ and $\mathcal{R}_{NL}$ denote a set of linear terms and non-linear terms, respectively. Note, $\mathcal{R}_L \cap \mathcal{R}_{NL} = \emptyset$. Let $\mathcal{R}_{NL} = \{p_1, \cdots, p_i, \cdots, p_u\} \cup \{q_1, \cdots, q_j, \cdots, q_w\}$, where p_i is a multiplication term, q_j is a division term, u and w are the number of multiplication and division terms, respectively.

Let $\widehat{\phi}$ denote a CORDIC approximation of ϕ, with $\widehat{\mathcal{B}}$, $\widehat{\mathcal{Z}}$, and $\widehat{\mathcal{R}}$ denoting the set of all *bool-expr*, *int-expr*, and *real-expr*, respectively in $\widehat{\phi}$. Steps to obtain $\widehat{\phi}$ are as follows:

1) Include all the linear terms of ϕ, i.e., $\widehat{\mathcal{B}} \leftarrow \mathcal{B}$, $\widehat{\mathcal{Z}} \leftarrow \mathcal{Z}$, $\widehat{\mathcal{R}} \leftarrow \mathcal{R}_L$.
2) For each multiplication term $p_i = s_i \times t_i \in \mathcal{R}_{NL}$, introduce *real-var*s M_{t_i}, i.e., $\widehat{\mathcal{R}} \leftarrow \widehat{\mathcal{R}} \cup \{M_{t_i}\}$. Update $\widehat{\mathcal{R}}$ and $\widehat{\mathcal{B}}$ with the Boolean and linear terms, arising from the CORDIC structure $M_{p_i} = cordicMult(s_i, M_{t_i}) + ec_i$, where $|ec_i| \leq (s \cdot 2^{-(n-1)})$ as given by (24).
3) Similar to step 2), for each division term $q_j = s_j/t_j \in \mathcal{R}_{NL}$, introduce *real-var*s M_{s_j}, M_{t_j}, i.e., $\widehat{\mathcal{R}} \leftarrow \widehat{\mathcal{R}} \cup \{M_{s_j}, M_{t_j}\}$. Update $\widehat{\mathcal{R}}$ and $\widehat{\mathcal{B}}$ with the Boolean terms and linear terms arising from the CORDIC structure $M_{q_j} = cordicDiv(M_{s_j}, M_{t_j}) + ec_j$, where $|ec_j| \leq 2^{-(n-1)}$ as given by (25).

Note, the Boolean-terms (in steps 2 and 3) arise due to ITE operators. Further, $\widehat{\mathcal{R}}$ contains only linear terms, including $\mathcal{R}_L$ and those arising from CORDIC structure and error correction.

Lemma 1: If ϕ is satisfiable with σ valuation, then there exists a satisfying valuation $\widehat{\sigma}$ such that $\forall_{b \in \mathcal{B}}\ \widehat{\sigma}(b) = \sigma(b)$, $\forall_{z \in \mathcal{Z}}\ \widehat{\sigma}(z) = \sigma(z)$, and $\forall_{r \in \mathcal{R}_L}\ \widehat{\sigma}(r) = \sigma(r)$.

D. Encoding ϕ to $\widehat{\phi}$ with Interval Bounds

Consider ϕ with interval bounds on non-linear terms. Specifically, for each multiplication term $p_i = s_i \times t_i$, define interval bound constraint IB_{p_i} as follows:

$$IB_{p_i} \stackrel{def}{=} \begin{cases} 2^{E_{t_i}} \leq |t_i| \leq 2^{E_{t_i}+1} & E_{t_i} \geq -(m-n) \\ |t_i| \leq 2^{-(m-n)} & otherwise \end{cases}$$

Similarly, for each division term $q_j = s_j/t_j$, define interval bound constraints IB_{qn_j} and IB_{qd_j} as follows:

$$IB_{qn_j} \stackrel{def}{=} \begin{cases} 2^{E_{t_i}} \leq |s_j| \leq 2^{E_{s_j}+1} & E_{s_j} - E_{t_j} \geq -(m-n) \\ |s_j| \leq 2^{-(m-n)} & otherwise \end{cases}$$

$$IB_{qd_j} \stackrel{def}{=} 2^{E_{t_j}} \leq |t_j| \leq 2^{E_{t_j}+1}$$

Let IB denote the conjunction of all interval bound constraints corresponding to multiplication and division terms.

$$IB \stackrel{def}{=} \bigwedge_i IB_{p_i} \wedge \bigwedge_j (IB_{qn_j} \wedge IB_{qd_j}) \tag{31}$$

Lemma 2: If ϕ is satisfiable with σ valuation, then there exists IB such that $\phi \wedge IB$ is satisfiable.

Given IB, we define $\widehat{\phi}|_{NC}$ (read as constraining $\widehat{\phi}$ with NC) as follows:

$$\widehat{\phi}|_{NC} \stackrel{def}{=} \widehat{\phi} \wedge \underbrace{\bigwedge_i NC_{p_i} \wedge \bigwedge_j (NC_{qn_j} \wedge NC_{qd_j})}_{NC} \tag{32}$$

where $NC_{p_i}, NC_{qn_j}, NC_{qd_j}$ are given by (26-30).

Lemma 3: Given IB as defined above (31), $\phi \wedge IB \Longrightarrow_{SAT} \widehat{\phi}|_{NC} \wedge IB$, i.e., the encoding is sound.

Proof. Assume $\phi \wedge IB$ is satisfiable for some σ valuation (Lemma 2). We show the satisfiability of $\widehat{\phi}|_{NC} \wedge IB$ by constructing a valuation of $\widehat{\sigma}$, that is also a model for $\widehat{\phi}|_{NC} \wedge IB$. Using Lemma 1, we obtain $\forall b \in \mathcal{B}\ \widehat{\sigma}(b) := \sigma(b)$, $\forall z \in \mathcal{Z}\ \widehat{\sigma}(z) := \sigma(z)$, $\forall r \in \mathcal{R}_L\ \widehat{\sigma}(r) := \sigma(r)$. We now show that there exists valuation $\widehat{\sigma}$ for every $r \in \widehat{\mathcal{R}} \backslash \mathcal{R}_L$ so that $NC \wedge IB$ is satisfiable.

Consider $NC_{p_i} \wedge IB_{p_i}$. We assign values M_{t_i} and M_{p_i} as follows:

- $\widehat{\sigma}(M_{t_i}) := \sigma(t_i) \cdot 2^{-E_{t_i}}$,
- $\widehat{\sigma}(M_{p_i}) := \sigma(p_i) \cdot 2^{-E_{t_i}}$.

From (14), we have to satisfy the following equation $M_{p_i} = cordicMult(s_i, M_{t_i}) + e_i \times s \cdot 2^{-(n-1)}$ with *real-var* $e_i \in [-1, 1]$. We assign $\widehat{\sigma}(e_i)$ as follows:

$$\widehat{\sigma}(e_i) := \frac{(\widehat{\sigma}(M_{p_i}) - cordicMult(\sigma(s_i), \widehat{\sigma}(M_{t_i})))}{(2^{-(n-1)} \cdot \sigma(s_i))} \tag{33}$$

As per (12), $|s_i \times M_{t_i} - cordicMult(s, M_{t_i})| \leq |s_i| \cdot 2^{-(n-1)}$ for $|M_{t_i}| < 2$; therefore, $|\widehat{\sigma}(e_i)| \leq 1$.

Similarly, consider $NC_{qn_j} \wedge NC_{qd_j} \wedge IB_{qn_j} \wedge IB_{qd_j}$. We assign M_{s_j}, M_{t_j}, and M_{q_j} as follows:

- $\widehat{\sigma}(M_{s_j}) := \sigma(s_j) \cdot 2^{-E_{s_j}}$,
- $\widehat{\sigma}(M_{t_j}) := \sigma(t_j) \cdot 2^{-E_{t_j}}$, and
- $\widehat{\sigma}(M_{q_j}) := \sigma(q_j) \cdot 2^{E_{t_j} - E_{s_j}}$.

From (21), we have to satisfy the following equation $M_{q_j} = cordicDiv(M_{s_j}, M_{t_j}) + e_j \cdot 2^{-(n-1)}$, with *real-var* $e_j \in [-1, 1]$. We assign $\widehat{\sigma}(e_j)$ as follows:

$$\widehat{\sigma}(e_j) := (\widehat{\sigma}(M_{q_j}) - cordicDiv(\sigma(s_j), \widehat{\sigma}(M_{t_j}))) \cdot 2^{(n-1)} \tag{34}$$

As per (19), $|M_{s_j}/M_{t_j} - cordicDiv(M_{s_j}, M_{t_j})| \leq 2^{-(n-1)}$ for $1 \leq |M_{t_j}| < 2$ and $|M_{q_j}|, |M_{s_j}| < 2$; and therefore, $|\widehat{\sigma}(e_j)| \leq 1$. Thus, we obtain a satisfying assignment $\widehat{\phi}$ for $\widehat{\phi}|_{NC} \wedge IB$. □

Theorem 1: $\phi \Longrightarrow_{SAT} \exists_{IB} \widehat{\phi}|_{NC} \wedge IB$

Proof. Follows from Lemmas 2 and 3. □

V. Numerical Accuracy

Given a satisfying solution for $\widehat{\phi}|_{NC} \wedge IB$, we derive error tolerances on each non-linear operation for given m, n, and bounded input/output domain.

Consider a multiplier, $p_i = s_i \times t_i$, with $t_i = M_{t_i} \cdot 2^{E_{t_i}}$. We have $M_{p_i} = cordicMult(s_i, M_{t_i}) + e_i \times s \cdot 2^{-(n-1)}$, and $p_i = M_{p_i} \cdot 2^{E_{t_i}}$. Using (15,16), we obtain absolute error err^M_{abs} and relative error err^M_{rel} as shown in Figure 2. Note, for $E_{t_i} \geq -(m-n)$, we have $1 \leq |\widehat{\sigma}(M_{t_i})| < 2$ by (26), otherwise, we have $|\widehat{\sigma}(M_{t_i})| < 2$ by (27).

Similarly, consider a division operation, $q_j = s_j/t_j$, with $s_j = M_s \cdot 2^{E_{s_j}}$, and $t_j = M_t \cdot 2^{E_{t_j}}$. We have $M_{q_j} = cordicDiv(M_{s_j}, M_{t_j}) + e_j \cdot 2^{-(n-1)}$, and $q_j = M_{q_j} \cdot 2^{E_{s_j} - E_{t_j}}$. Using (22,23), we obtain absolute error err^D_{abs} and relative error err^D_{rel} as shown in Figure 2. Note, for $E_{s_j} - E_{t_j} \geq -(m-n)$, we have $1 \leq |\widehat{\sigma}(M_{t_j})| < 2$ and $1 \leq |\widehat{\sigma}(M_{s_j})| < 2$, and $1/2 \leq |\widehat{\sigma}(M_{q_j})| < 2$, by virtue of (28-29); otherwise, we have $|\widehat{\sigma}(M_{s_j})| < 2$, $1 \leq |\widehat{\sigma}(M_{t_j})| < 2$, and $|\widehat{\sigma}(M_{q_j})| < 2$, by virtue of (29-30).

Remark: As long as the inputs are normalized (i.e., by choosing large m) the relative error tolerances are independent on the input values (but dependent on n). On the other hand, when the output value approaches zero, the absolute error is dependent on m (with potentially large relative error). Such error tolerances are generally accepted by standard such as IEEE754-2008 [18]. This feature is meant to provide a slowing of the precision loss due to cancelation effects around zero.

$$err^M_{abs} \leq \begin{cases} |\widehat{\sigma}(p_i) - \sigma(p_i)| \leq 2^{-(n-2)+E_{t_i}} \cdot |s| & E_{t_i} \geq -(m-n) \\ |\widehat{\sigma}(p_i) - \sigma(p_i)| \leq 2^{-(m-2)} \cdot |s_i| & otherwise \end{cases}$$

$$err^M_{rel} \leq \begin{cases} \frac{|\widehat{\sigma}(p_i) - \sigma(p_i)|}{|\sigma(p_i)|} \leq 2^{-(n-2)} & E_{t_i} \geq -(m-n) \\ \frac{|\widehat{\sigma}(p_i) - \sigma(p_i)|}{|\sigma(p_i)|} \leq \frac{2^{-(n-2)}}{|\sigma(M_{t_i})|} & otherwise \end{cases}$$

$$err^D_{abs} \leq \begin{cases} |\widehat{\sigma}(q_j) - \sigma(q_j)| \leq \frac{2^{-(n-2)}}{-(E_{s_j} - E_{t_j})} & E_{s_j} - E_{t_j} \geq -(m-n) \\ |\widehat{\sigma}(q_j) - \sigma(q_j)| \leq 2^{-(m-2)} & otherwise \end{cases}$$

$$err^D_{rel} \leq \begin{cases} \frac{|\widehat{\sigma}(q_j) - \sigma(q_j)|}{|\sigma(q_j)|} \leq 2^{-n+3} & E_{s_j} - E_{t_j} \geq -(m-n) \\ \frac{|\widehat{\sigma}(q_j) - \sigma(q_j)|}{|\sigma(q_j)|} \leq \frac{2^{-(n-2)}}{|\sigma(M_{q_j})|} & otherwise \end{cases}$$

σ is the satisfying solution of ϕ.
$\widehat{\sigma}$ is the satisfying solution of $\widehat{\phi}$.

Fig. 2. Bounds on absolute and relative errors

VI. $\mathcal{DISE}$: DPLL-BASED INTERVAL SEARCH ENGINE

Before we present the search strategies and the search procedure, we describe the input problem and notations used.

A. Problem Description

Assume we are given a non-linear decision problem ϕ with interval bounds on non-linear terms. We refer to these interval bounds as *external* and differentiate them from interval bounds IB (31), which we refer to as *internal*. Specifically, for each multiplication term $p_i = s_i \times t_i$, let the external interval bound constraint xIB_{p_i} corresponds to finite interval bounds such that $t_i \in [\underline{t_i}, \overline{t_i}]$. Similarly, for each division term $q_j = s_j / t_j$, let the external interval bound constraint xIB_{qn_j} and xIB_{qd_j} correspond to given finite interval bounds such that $s_j \in [\underline{s_j}, \overline{s_j}]$ and $t_j \in [\underline{t_j}, \overline{t_j}]$, respectively.

Let xIB denote conjunction of all the external interval bound constraints corresponding to each multiplication and division.

$$xIB \stackrel{def}{=} \bigwedge_i xIB_{p_i} \wedge \bigwedge_j (xIB_{qn_j} \wedge xIB_{qd_j}) \tag{35}$$

$\widehat{\phi}$ denotes SMT($\mathcal{LA}$) formula including the linear-part of ϕ and the translated part of non-linear expression, obtained after CORDIC approximation as described in Section IV-D. The procedure $\mathcal{DISE}$ takes the formula $\widehat{\phi} \wedge xIB$ and outputs $UNSAT$, or SAT with a satisfying model. It invokes a SMT($\mathcal{LA}$)-solver for consistency checks of $\psi \stackrel{def}{=} \widehat{\phi} \wedge xIB \wedge NC \wedge IB$, where IB is given by (31) and NC is given by (32). It incrementally asserts/retracts each interval constraints, as *guided* by the search strategies described next. Note, finite external bounds xIB are needed for termination of our search procedure, as described later.

B. Search Strategies

To allow a formula to be solved incrementally, we *do not want to re-encode* the entire formula when some interval bounds change. Re-encoding of the formula looses all the information that were learnt in the previous run. We describe two guiding mechanisms i.e., *theory suggestion* and *theory learning*, which will be the basic search strategies in $\mathcal{DISE}$.

Consider a SMT($\mathcal{LA}$) formula $\psi = \psi_H \wedge \psi_S$, where we would like the SMT($\mathcal{LA}$)-solver to satisfy ψ_H, but to choose not to satisfy ψ_S; and report satisfiable or unsatisfiable based on the check for ψ_H alone. We call ψ_H as *hard constraints*, and ψ_S as *soft constraints*. Typically, an SMT($\mathcal{LA}$)-solver (such as [19]) provides such checking capability, where ψ_S constraints can be asserted with some finite *weights*, and ψ_H can be asserted with infinite weights, and the solver finds a weighted model. Based on the satisfiability result on ψ, we have the following guidance strategies:

Theory Suggestion: If ψ is returned satisfiable, ψ_H is satisfiable, but ψ_S may not be satisfiable. In such a case, we use the model values of variables in ψ_S to analyze which constraints in ψ_S are unsatisfiable/satisfiable. We use that information to "suggest" a new set of intervals and modify the constraints ψ_S accordingly. We use this suggestion-based mechanism in our interval search to guide our decision on a combination of interval bounds that will constitute ψ_S.

Theory Learning: If ψ is returned unsatisfiable, ψ_H is also unsatisfiable. In such a case, we use the unsat core of ψ_H to identify which constraints in ψ_H were the causes for unsatisfiability. Typically, an SMT($\mathcal{LA}$)-solver would provide such a sufficient set of constraints using unsat core analysis. We use this information to "learn" a blocking clause and update the constraints ψ_H accordingly. We use this learning-based mechanism to guide our non-chronological backtracking.

C. Interval Search Procedure

As noted above, we *do not want to re-encode* the entire formula when some bounds change. Therefore, we choose *not* to change the constraints $\widehat{\phi} \wedge xIB$. We assert them as *hard* constraints during the entire search process. However, as we add/remove the individual constraints of NC and IB, we change some of the constraints IB and NC from *soft* to *hard* (in a decision step) and *hard* to *soft* (in a backtracking step). Note, our goal is to find some interval bounds IB that will satisfy the given formula, or to show that no such satisfying interval bounds exists (in which case, the formula $\phi \wedge xIB$ is unsatisfiable).

Let Ω be the set of possible interval bounds constraints used as decision candidates, i.e., $\Omega = \{IB_{p_1} \wedge NC_{p_1}, \cdots, IB_{p_u} \wedge NC_{p_u}, IB_{qn_1} \wedge NC_{qn_1} \cdots IB_{qn_w} \wedge NC_{qn_w}, IB_{qd_1} \wedge NC_{qd_1} \cdots IB_{qd_w} \wedge NC_{qd_w}\}$. Note, each $\omega(r, \rho) \in \Omega$ is a Boolean predicate on a real term r, and an interval bound ρ.

Let $\mathcal{C} \subseteq \Omega$ be a set of interval bound constraints that have been chosen (i.e. decided) up to the current decision level. The constraints in the set are asserted as hard constraints. We also refer the set $\mathcal{C}$ as the *committed* set. Let $\mathcal{U} = \Omega \backslash \mathcal{C}$, denotes the uncommitted set. The constraints in the set are asserted as soft constraints. Let $\mathcal{E}$ denote the learned constraints such that $\neg\mathcal{E}$ corresponds to the interval bounds with no solution, as established in the previous searches. These constraints are asserted as hard constraints. The formula ψ constructed as such is shown below.

$$\psi = \underbrace{\widehat{\phi} \wedge xIB}_{hard} \wedge \underbrace{\bigwedge_{\omega \in \mathcal{C}} \omega}_{hard} \wedge \underbrace{\bigwedge_{\nu \in \mathcal{U}} \nu}_{soft} \wedge \underbrace{\mathcal{E}}_{hard} \quad (36)$$

$\mathcal{DISE}$ proceeds in a DPLL-style with backtracking as follows: Initially, $\mathcal{C} = \emptyset$, $\mathcal{U} = \Omega$, $\mathcal{E} = false$. At a particular decision level (starting from 0), it selects a soft constraint $\nu \in \mathcal{U}$, and asserts it as a hard constraint. Based on the results of SMT($\mathcal{LA}$)-solver on ψ, $\mathcal{DISE}$ makes one of the following actions.

Case 1. If ψ is satisfiable and $\mathcal{U} = \emptyset$, it returns SAT.

Case 2. If ψ is unsatisfiable and $\mathcal{C} = \emptyset$, it returns UNSAT.

Case 3. If ψ is satisfiable with weighted model $\widehat{\sigma}$, and $\mathcal{U} \neq \emptyset$, it does the following:

- If all $\nu(r, \rho) \in \mathcal{U}$ are $true$, it returns SAT; otherwise, for each $\nu(r, \rho) \in \mathcal{U}$ such that ν is $false$, (a) the interval bound of corresponding variable r is adjusted to ρ' such that $\sigma(r) \in \rho'$ and $\nu(r, \rho')$ is satisfied (ref. Section IV-D), and (b) $\mathcal{U} \leftarrow \mathcal{U} \backslash \{\nu(r, \rho)\} \cup \{\nu(r, \rho')\}$.
- Picks some $\nu(r, \rho) \in \mathcal{U}$ as the next decision; Updates $\mathcal{C}$,$\mathcal{U}$ i.e., $\mathcal{C} \leftarrow \mathcal{C} \cup \{\nu\}$, and $\mathcal{U} \leftarrow \mathcal{U} \backslash \{\nu\}$

Case 4. If ψ is unsatisfiable and $\mathcal{C} \neq \emptyset$, it does the following:

- It obtains a set $\mathcal{IC} \subseteq \mathcal{C}$ that is a sufficient set for unsatisfiability. It constructs a conjunction of infeasible constraints, i.e., $IC = \bigwedge_{\omega \in \mathcal{IC}} \omega$. It then updates the constraints set $\mathcal{E}$ as follows: $\mathcal{E} \leftarrow \mathcal{E} \wedge \neg IC$. Note, this partial combination of interval bounds will not be explored by the solver in the future.
- It backtracks (non-chronologically) to a previous decision level $bl = max\{dlevel(\omega) \mid \omega \in \mathcal{IC} \backslash \{\zeta\}\}$, where $dlevel(\omega)$ is the decision level of ω and ζ is the last decision in $\mathcal{IC}$. It updates $\mathcal{C}$ and $\mathcal{U}$: $\mathcal{C} \leftarrow \mathcal{C} \backslash \Theta, \mathcal{U} \leftarrow \mathcal{U} \cup \Theta$, where Θ is a set of decisions to skip-over i.e., $\Theta = \{\omega | dlevel(\omega) > bl\}$.

We present our overall decision procedure $\mathcal{CORD}$ using $\mathcal{DISE}$ in Algorithm 1. We use the procedure `CORDIC_Translate` to obtain CORDIC translation as described in Section III. We assume that the procedures indicated in lines 5—10 are supported in a SMT($\mathcal{LA}$)-solver to allow incremental formulation of sub-problems. We use a `Stack` for recording and backtracking purposes.

***Theorem** 2:* $\mathcal{CORD}$ always terminates, either with a correct unsatisfiability result, or with a satisfiability result that is correct within the given precision requirements.

Proof: Refer [20].

VII. Experiments

We implemented our $\mathcal{CORD}$ algorithm using the SMT solver *yices-1.0.20* [19]. Our experiments were conducted on a single threaded environment, on a workstation with 3.4GHz, 2GB of RAM running Linux. We used a time limit of one hour for each decision problem. All the benchmarks used are publicly available [20], [21].

In the first experiment, we evaluate the performance of $\mathcal{CORD}$ with various relative error tolerance (denoted as $2^{-\delta}$) on non-linear decision problems corresponding to numerical programs `Ex1-Ex3`. Each problem is generated at different depths (between 12 to 26) using SMT-based BMC [2], [22]

Algorithm 1 $\mathcal{CORD}$: CORDIC-based Decision Procedure

1: **input:** Non-linear decision problem ϕ with external interval bounds on non-linear inputs, i.e., $\phi \wedge xIB$.
2: **output:** $SAT/UNSAT$
3:
4: {Following procedures are supported by SMT($\mathcal{LA}$) solver}
5: {`SMT_Init`: Encode logical expressions to SMT($\mathcal{LA}$)}
6: {`SMT_Assert`: Assert constraints with infinite weights (i.e., hard constraints) that can be retracted later}
7: {`SMT_Assert_Weighted`: Assert constraints with finite weights (i.e., soft constraints) that can be retracted later}
8: {`SMT_Retract`: Retract previously asserted constraints}
9: {`SMT_Find_Weighted_Model`: Check the satisfiability of non-weighted constraints, and obtain a weighted model}
10: {`SMT_Unsat_Core`: Returns a sufficient subset of unsatisfying constraints}
11:
12: $\widehat{\phi}$:= `CORDIC_Translate`(ϕ) {Sec. III}
13: `SMT_Init`($\widehat{\phi} \wedge xIB$) {Encode to SMT($\mathcal{LA}$). Sec. IV}
14: {$\Omega \equiv$ the set of decision candidates}
15: $\mathcal{C} := \emptyset$ {Committed set $\subseteq \Omega$}
16: $\mathcal{U} := \Omega$ {Uncommitted set $\Omega \backslash \mathcal{C}$}
17: `Stack_Init()` {Initialize the decision stack}
18:
19: **for all** $\omega(r, \rho) \in \Omega$ **do**
20: $\rho = [1, 2)$ {Initial interval bound for non-linear input r}
21: `SMT_Assert_Weighted`($\omega(r, \rho), wt$) {$r \in \rho$}
22: **end for**
23:
24: {$\mathcal{DISE}$: DPLL-based Interval Search Engine}
25: **loop**
26: {Invoke SMT($\mathcal{LA}$)-solver}
27: is_sat := `SMT_Find_Weighted_Model()`
28:
29: {Case 1}
30: **if** (is_sat = $true$) and ($\mathcal{U} = \emptyset$) **then**
31: return SAT {All bounds are satisfied}
32: **end if**
33:
34: {Case 2}
35: **if** (is_sat = $false$) and ($\mathcal{C} = \emptyset$) **then**
36: return UNSAT {Unsatisfiable}
37: **end if**
38:
39: {Case 3: update bounds, decide}
40: **if** (is_sat = $true$) and ($\mathcal{U} \neq \emptyset$) **then**
41: Let $\widehat{\sigma}$ be the weighted model found
42: num_update := 0 {number of interval updates}
43: **for all** $\nu(r, \rho) \in \mathcal{U}$ s.t $\widehat{\sigma}(\nu(r, \rho)) = false$ **do**
44: Select ρ' s.t $\widehat{\sigma}(\nu(r, \rho')) = true$ {$r \in \rho'$}
45: {$\mathcal{U} := \mathcal{U} \backslash \{\nu(r, \rho)\} \cup \{\nu(r, \rho')\}$}
46: `SMT_Retract`($\nu(r, \rho)$) {retract old bound}
47: `SMT_Assert_Weighted`($\nu(r, \rho')$,wt)
48: num_update++; {increase the update count}
49: **end for**
50: **if** (num_update = 0) **then**
51: return SAT {all model values are satisfying}
52: **end if**
53: Pick $\omega(r, \rho) \in \mathcal{U}$ {Decide on a variable r and interval ρ}
54: $\mathcal{C} := \mathcal{C} \cup \{\omega\}$, $\mathcal{U} := \mathcal{U} \backslash \{\omega\}$. {update $\mathcal{C}$ and $\mathcal{U}$}
55: `Stack_push`(ω) {Store the last decision}
56: **end if**
57:
58: {Case 4: backtrack, add blocking constraint}
59: **if** (is_sat = $false$) and ($\mathcal{C} \neq \emptyset$) **then**
60: $\mathcal{IC}$:= `SMT_Unsat_Core`($\mathcal{C}$) {Unsat core analysis}
61: $IC := \bigwedge_{\omega \in \mathcal{IC}} \omega(r, \rho)$ {infeasible combination}
62: `SMT_Assert`($\neg IC$) {block infeasible region}
63: Let ζ be the last decision among $\omega \in \mathcal{IC}$
64: Let bl be the maximum decision level among $\omega \in \mathcal{IC} \backslash \{\zeta\}$
65: $\Theta = \{\omega \mid dlevel(\omega) > bl\}$ {decisions to skip-over}
66: $\mathcal{U} := \mathcal{U} \cup \Theta$, $\mathcal{C} := \mathcal{C} \backslash \Theta$ {backtrack}
67: `Stack_pop`(bl) {backjump to bl}
68: **end if**
69: **end loop**

TABLE I
PERFORMANCE EVALUATION OF $\mathcal{CORD}$

Ex (S/U)(k)	$\delta=6$			$\delta=8$			$\delta=10$			$\delta=12$		
	T	D	B	T	D	B	T	D	B	T	D	B
Ex1 with 4/8/12 **m**ultipliers with **s**at/**u**nsat result; $n=\delta+2$												
4m(S)(12)	2.4	4	0	2.3	4	0	6.6	4	0	3	4	0
4m(U)(12)	1.3	4	4	2.8	4	4	6.8	4	4	2.6	4	4
8m(S)(19)	155	113	105	62	15	7	648	113	105	569	43	35
8m(U)(19)	298	225	225	432	225	225	1249	225	225	2193	232	232
12m(S)(26)	524	1554	1566	962	577	589	978	171	183	1328	0	12
Ex2 with 4/8/12 **m**ultipliers with **s**at/**u**nsat result; $n=\delta+2$												
4m(S)(12)	1.5	4	0	2.6	4	0	2.9	4	0	2.8	4	0
4m(U)(12)	1.5	4	4	2	4	4	2.5	4	4	3.52	4	4
8m(S)(19)	33	20	12	28.6	8	0	203	1569	1561	225.3	33	25
8m(U)(19)	49.6	117	117	83.8	80	80	175	37	37	461	67	67
12m(S)(26)	101	1623	1611	191	42	30	488	100	88	2396	43	31
Ex3 with 4 **d**ivision with **s**at/**u**nsat result; $n=\delta+3$												
4d(S)(12)	5.8	10	4	131	122	79	11.6	16	8	64.9	32	19
4d(U)(12)	55.6	188	126	126.1	188	126	308.8	188	126	465.5	188	126

Relative error $2^{-\delta}$ **T**: Time used (in sec) **D/B**: # of decisions/backtracks

TABLE II
COMPARISON OF iSAT AND $\mathcal{CORD}$

Ex (S/U)	# mult+ div	δ	CORD			iSAT [14]	
			n	S/U	T(sec)	S/U	T(sec)
e1(U)	1+0	10	12	U	2.5	U	0
e2(S)	1+0	13	15	S	2.7	U?	0
e3(U)	1+0	15	17	U	0.03	S?	0
e4(S)	1+0	20	22	S	4.3	S	1231
NS: Checking numerical stability							
s1(U)	1+0	18	20	U	4.4	?	Mem Out
s2(U)	1+0	18	20	U	4.7	?	Mem Out
s3(S)	1+0	18	20	S	55	?	Mem Out
s4(S)	1+1	18	21	S	770	?	Mem Out

on a model derived from the numerical programs [23]. The results are shown in Table I. Column 1 presents the decision problem `<i>m/d(S/U)<k>`, wherein `<i>` denotes the number of multiplication(`m`)/division(`d`), `S/U` denote whether it is a satisfiable/unsatisfiable problem (irrespective of error bounds), and `<k>` denotes the BMC depth of k. Columns 2—4, 5—7, and 8—9 present results for different values of δ. Especially, Column 2 presents time taken `T` (in sec), Columns 3 and 4 present number of decisions `D` and backtracks `B` in $\mathcal{DISE}$, respectively. For this set, we used $m=64$. To meet the error tolerance, we select n (n is the number of CORDIC steps) as follows: for decision problems with multiplication only, we use $n=\delta+2$; otherwise, (in presence of division), we use $n=\delta+3$ (Figure 2). With increasing δ, the performance is affected due to increase in the sizes of the CORDIC structure. *We could not present any result with iSAT on this set, as it memory outs on all.*

In the second experiment, we compared $\mathcal{CORD}$ with iSAT on two sets of challenging benchmarks `e1-e4`, and `s1-s4`. The latter set corresponds to decision problems arising from analysis of numerical stability of programs [24]. We present the results in Table II. Column 1 presents the example with `(S/U)` denoting an expected result, i.e., satisfiable or unsatisfiable (irrespective of error bounds), respectively. Column 2 presents the total number of multiplications and divisions. Column 3 presents the required precision (relative error tolerance), $2^{-\delta}$. Column 4–6 present results for $\mathcal{CORD}$, with `n` denoting the number of CORDIC steps to meet the required precision; Time used (in sec) `T`, and sat or unsat `(S/U)`, respectively. Similar results are presented for iSAT in Columns 7—8. Similar to the first set, we used $m=64$. Note, for `e2-e3`, iSAT gives spurious results, and for the set `s1-s4`, it memory outs.

VIII. CONCLUSION

We presented a novel decision procedure $\mathcal{CORD}$ for linear/non-linear constraints using CORDIC algorithms. We first obtain a safe translation to SMT($\mathcal{LA}$) formula, accounting for the inaccuracies in a CORDIC approximation. We then perform a DPLL-style interval search ($\mathcal{DISE}$) on the encoded formula using an off-the-shelf SMT($\mathcal{LA}$)-solver. We use theory suggestion and theory learning for an effective guidance of interval search. We also presented upper bounds on absolute and relative errors due to CORDIC approximation in a satisfying solution. In future, we plan to extend the presented approach to support transcendental functions. We conclude that such an approach is crucial for both hardware and software verification methodologies in practice.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, 1999.

[2] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proc. of ICCAD*, 2006.

[3] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proc. of CAV*, 2006.

[4] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propogation and its application to difference logic. In *Proc. of CAV*, 2005.

[5] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. V. Rossum, M. Schulz, and R. Sebastiani. The MathSAT 3 System. In *Proc. of CADE*, 2005.

[6] A. Bauer, M. Pister, and M. Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Proc. of DATE*, 2007.

[7] M. Franzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal of Satisfiability, Boolean Modeling and Computation*, 2007.

[8] J. Volder. The CORDIC trigonometric computing technique. In *IRE Trans. Electron. Comput., vol. EC-8, no. 3, pp. 33g-334*, 1959.

[9] J. S. Walther. A unified algorithm for elementary functions. In *AFIPS Spring Joint Computer Conf*, 1971.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 1962.

[11] SMT Research Community. SMT-LIB. *http://combination.cs.iowa.edu/smtlib/*.

[12] A. Wachter and L. T. Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. In *SIAM Journal of Control and Optimization*, 2005.

[13] IPOPT team. Ipopt homepage. *https://projects.coin-or.org/Ipopt*.

[14] HySAT team. Hysat team. *http://hysat.informatik.uni-oldenburg.de/*.

[15] M. K. Ganai. Efficient decision procedure for bounded integer non-linear operations using smt(lia). In *Haifa Verification Conference*, 2008.

[16] A. Oliveras G. Faure, R. Nieuwenhuis and E. R.-Carbonell. Sat modulo the theory of linear arithmetic: Exact, inexact and commercial solvers. In *Theory and Applications of Satisfiability Testing*, 2008.

[17] Y. H. Hu. The quantization effects of the CORDIC algorithm. *IEEE Transactions on Signal Processing*, April 1992.

[18] IEEE 754-2008. IEEE standard for floating-point arithmetic. *http://ieeeexplore.ieee.org/servlet/opac?punumber=4610933*.

[19] SRI. Yices: An SMT solver. *http://fm.csl.sri.com/yices*.

[20] M. K. Ganai. Conference Notes. *http://www.nec-labs.com/∼malay/notes.htm*.

[21] System Analysis and Verification Team. NECLA SAV Benchmarks. *http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php*.

[22] M. Franzle and C. Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. In *Formal Method in System Design*, 2006.

[23] F. Ivančić, M. K. Ganai, S. Sankaranarayanan, and A. Gupta. Sofware Model Checking the Precision of Floating-Point Programs. In *Under Submission*, 2009.

[24] E. Goubault. Static analyses of the precision of floating-point operations. In *Proc. of SAS*, 2001.

Mixed Abstractions for Floating-Point Arithmetic

Angelo Brillout
Computer Systems Institute, ETH Zurich

Daniel Kroening and Thomas Wahl
Oxford University Computing Laboratory

***Abstract*—Floating-point arithmetic is essential for many embedded and safety-critical systems, such as in the avionics industry. Inaccuracies in floating-point calculations can cause subtle changes of the control flow, potentially leading to disastrous errors. In this paper, we present a simple and general, yet powerful framework for building abstractions from formulas, and instantiate this framework to a bit-accurate, sound and complete decision procedure for IEEE-compliant binary floating-point arithmetic. Our procedure benefits in practice from its ability to flexibly harness both over- and underapproximations in the abstraction process. We demonstrate the potency of the procedure for the formal analysis of floating-point software.**

I. Introduction

Embedded systems are typically controlled by software that conceptually manipulates real-valued quantities, for instance measurements of environment data. Such quantities are stored in a computer as *floating-point numbers*. As only few real numbers can be encoded in this format, values must generally be *rounded* to some nearby floating-point number.

Compared to a computation with infinite precision, rounding can influence program behavior in multiple ways. The deviation caused by rounding can lead to unintuitive results, such as in a non-associative addition operation. Worse, the deviation can accumulate and eventually change the control flow of the program. Implementations of floating-point algorithms can be sensitive to very small variations in input. Bugs caused by such rounding errors are therefore often hard to reproduce and to test for, and have been referred to as "Heisenbugs" [1]. If undetected, they can have tragic consequences, as embedded devices are used in many mobile and ubiquitous computing environments. A prominent example is the Ariane 5 disaster, caused by an out-of-bounds 64-bit floating-point conversion. The indisputable need for reliability in embedded applications calls for precise and rigorous formal analysis methods.

Programs with floating-point arithmetic have been addressed in the past in various ways. In *abstract interpretation* [2], the program is (partially) executed on an abstract domain, such as real intervals. The generated transformations may, however, turn out too coarse for definite decisions on the given properties. *Proof assistants* are tools that prove theorems about programs (involving floating-point arithmetic) under human guidance. This guide, unfortunately, must be highly skilled to direct the tool towards a proof. Both abstract interpretation and theorem proving often lack the ability to generate counterexamples for invalid properties, which is essential for debugging and for the high-impact field of automated test-vector generation.

In this paper, we present a precise and sound decision procedure for (binary) floating-point arithmetic for the automatic analysis of software. A principal way of achieving this is to encode floating-point operations as functions on bit-vectors, and relying on efficient solvers for bit-vector logic, for instance those based on "bit-flattening" and subsequent SAT-solving. Unfortunately, this approach has proven to be intractable in practice, simply because it results in very large and hard-to-solve SAT instances, as we will illustrate.

A common solution to address this problem is to use *approximations* of formulas. Particularly, an overapproximation $\overline{\phi}$ simplifies the formula ϕ in a way that preserves all satisfying assignments; thus, unsatisfiability of $\overline{\phi}$ implies unsatisfiability of ϕ. Analogously, an underapproximation $\underline{\phi}$ simplifies ϕ in a way that removes satisfying assignments; thus, any satisfying assignment to $\underline{\phi}$ can be adjusted to one for ϕ.

Classical counterexample-guided abstraction refinement (*CEGAR*) [3] relies on overapproximations, which are refined if spurious counterexamples (in the form of spurious satisfying assignments) are encountered. In [4], a decision procedure for bit-vector arithmetic is presented that employs both types of approximations, although in a fixed alternation schedule. Such approaches are too rigid for floating-point arithmetic, as some formulas do not permit effective overapproximations, while others do not permit effective underapproximations.

In this paper, we propose a new abstraction method for checking the satisfiability of a floating-point formula ϕ. Our algorithm permits a mixed sequence S of both over- and underapproximating transformations. The formula ψ resulting from applying S to ϕ is in general neither an over- nor an underapproximation of ϕ. Our algorithm stops whenever (i) the simplified formula ψ permits a satisfying assignment that satisfies ϕ, too, or (ii) ψ is unsatisfiable and permits a resolution proof that is also a valid proof for ϕ. If neither of the opportunities (i) and (ii) applies, S needs to be refined. If ψ was found to be spuriously satisfiable, the algorithm removes an overapproximating transformation from S, otherwise an underapproximating transformation.

Our algorithm can be seen as a framework for a class of abstraction-based procedures to check the satisfiability of formulas in some logic: different choices of the transformation sequence S result in different instances of our framework. For example, the work presented in [4] is an instance where S strictly alternates between over- and underapproximations.

This research is supported by the the Toyota Motor Corporation, by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539, by the EU FP7 STREP MOGENTES (project ID ICT-216679), and by the EPSRC project EP/G026254/1.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

CEGAR is an instance where S contains only overapproximations. Our paper generalizes these methods in a way that permits a choice of approximations based on their effectiveness to simplify the input formula. Termination of any algorithm based on our framework is guaranteed as long as the sequence S can be shown to be depleted eventually.

We demonstrate the utility of the procedure on decision problems arising in bounded model checking (BMC) [5] of ANSI-C programs.

II. Preliminaries: Floating-Point Arithmetic

A. The IEEE floating-point format

The binary *floating-point format* is used to represent real numbers in a computer. Specifically, the triple consisting of a *sign* $s \in \{0,1\}$, an integer-valued *exponent* e, and a rational-valued *mantissa* m represents the floating-point number $(-1)^s \cdot m \cdot 2^e$. According to the IEEE standard 754, the three components are encoded using bit-vectors, resulting in the partitioned representation of a floating-point number shown in Figure 1.

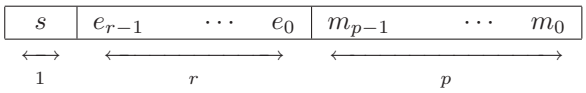


Fig. 1. The three fields of an IEEE-754 floating-point number

The sign bit s directly represents the sign of the floating-point number. The following two bit-vector fields are interpreted as follows:

- The bit field $\bar{e} := e_{r-1} \dots e_0$ encodes the integral exponent e as a binary number.
- Together with the hidden bit, the bit field $\bar{m} := m_{p-1} \dots m_0$ encodes the fractional value of the mantissa m; the representation ensures that $0 \le m < 2$. The hidden bit is derived from $\bar{e}$, and is used to distinguish normal and denormal numbers.

The widths r and p of the second and third bit fields in Figure 1 are called the *range* and the *precision* of the representation. The IEEE standard 754 defines two types of floating-point numbers: the *single* format with $(r,p) = (8,23)$, and the *double* format with $(r,p) = (11,52)$.

Unrepresentable real numbers are rounded, as we review in Section II-B. Numbers that are too large are represented using the symbols $-\infty$ and $+\infty$. The floating-point number NaN ("Not a Number") represents results of operations outside of real arithmetic, such as imaginary values. We call the floating-point numbers $\pm\infty$ and NaN *special*; they are represented using reserved patterns for exponent and mantissa.

In this paper, we manipulate floating-point formulas by varying the precision parameter p, while parameter r is fixed. We denote by $\mathbb{F}_p$ the set consisting of the floating-point numbers $(-1)^s \cdot m \cdot 2^e$ representable as a bit-vector with precision p, and the special numbers $\pm\infty$. With $\mathbb{R}^\infty := \mathbb{R} \cup \{\pm\infty\}$, we have $-\infty \le x \le +\infty$ for all $x \in \mathbb{R}^\infty$. Obviously, for $p' \le p$, we have $\mathbb{F}_{p'} \subseteq \mathbb{F}_p \subset \mathbb{R}^\infty$.

B. Floating-point arithmetic

The result of an operation $a \circ b$, for $\circ \in \{+,-,\times,/\}$, may not be representable in $\mathbb{F}_p$ even though a and b are in $\mathbb{F}_p$.[1] In such a case, an appropriate approximation is selected. For $x \in \mathbb{R}$, define the approximations $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$ as

$$\begin{aligned} \lfloor x \rfloor_p &:= \max\{f \in \mathbb{F}_p : f \le x\}\,, \quad \text{and} \\ \lceil x \rceil_p &:= \min\{f \in \mathbb{F}_p : f \ge x\}\,. \end{aligned}$$

The values $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$ are the two floating-point numbers in $\mathbb{F}_p$ nearest to x. There is no floating-point number strictly between $\lfloor x \rfloor_p$ and $\lceil x \rceil_p$. If x is larger than any non-special floating-point number in $\mathbb{F}_p$, then $\lceil x \rceil_p = +\infty$; analogously for $\lfloor x \rfloor_p$. The approximation values satisfy the following **nesting property**: for $p' \le p$, $\lfloor x \rfloor_{p'} \le \lfloor x \rfloor_p \le x \le \lceil x \rceil_p \le \lceil x \rceil_{p'}$.

Definition 1: A *rounding function* is a function $rd_p \colon \mathbb{R} \to \mathbb{F}_p$ such that, for all $x \in \mathbb{R}$, $rd_p(x) \in \{\lfloor x \rfloor_p, \lceil x \rceil_p\}$.

Specific rounding functions are also known as *rounding modes*. Two examples of rounding modes are *round-up* ($rd_p = \lceil \cdot \rceil_p$) and *round-down* ($rd_p = \lfloor \cdot \rfloor_p$).

The floating-point operators $\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}$ are defined as the *rounded* result of the corresponding real operators $\circ \in \{+,-,\times,/\}$:

Definition 2: For a given rounding function rd_p and an arithmetic operation $\circ \colon \mathbb{R}^2 \to \mathbb{R}$, the corresponding floating-point operation $\circledcirc_p \colon \mathbb{F}_p^2 \to \mathbb{F}_p$ is defined by

$$x \circledcirc_p y := rd_p(x \circ y)\,.$$

The IEEE standard 754 extends this definition to operations with special operands, e.g. $+\infty \oplus_p -\infty :=$ NaN. Note that, due to the rounding, associativity does not hold for floating-point operations, i.e., $(a \circledcirc_p b) \circledcirc_p c$ may differ from $a \circledcirc_p (b \circledcirc_p c)$.

Floating-point arithmetic (FPA) (with precision p) is the logic defined by the structure $\langle \mathbb{F}_p, =, \le, \circledcirc \rangle$. A *term* is an expression over arithmetic operations involving variables or constants over $\mathbb{F}_p$. We also allow conversions between terms with different precision. Given terms t_1, t_2, atoms in FPA are of the form $t_1 \bowtie t_2$ with $\bowtie \in \{=, \le\}$. The Boolean connectives $\wedge$, $\vee$, $\neg$ are used to construct formulas. We consider the combination of FPA with integer bit-vector arithmetic (BVA) and allow both semantic and bit-wise conversions between integer bit-vectors and floating-point bit-vectors. The goal of this work is a decision procedure that determines the satisfiability of FPA+BVA formulas.

III. Propositional Encodings of FPA Formulas

Given a circuit implementation of an IEEE-754 compliant floating-point unit (FPU), each floating-point operation can be modeled as a formula in propositional logic, as we illustrate below. This way, a formula in FPA can — in principle — be translated to an equisatisfiable formula in propositional logic and passed to a SAT-solver to check for satisfiability. This suggests a sound and complete decision procedure for FPA.

[1] For instance, the addition of the binary numbers $1.1 \cdot 2^0 \in \mathbb{F}_1$ and $1.0 \cdot 2^0 \in \mathbb{F}_1$ (1 bit fractional precision) results in $10.1 \cdot 2^0$, which is not representable with 1 bit fractional precision and a mantissa $m < 2$.

The bottleneck is of course the complexity of the resulting propositional formulas, as we demonstrate in the following. Our analysis also hints at sources for *approximating* these formulas in meaningful ways.

A. Addition and Subtraction

Figure 2 shows a high-level description of a floating-point adder/subtracter as implemented in most FPUs. An adder/subtracter is composed of three modules.

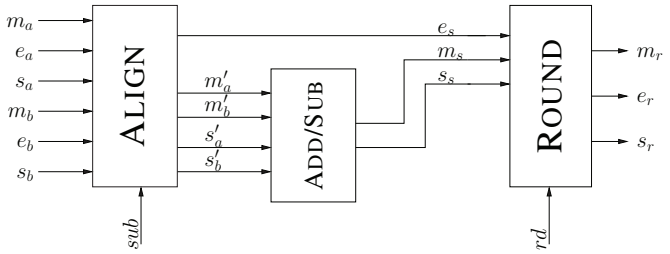


Fig. 2. High-level overview of a floating-point adder/subtracter.

- ALIGN. The mantissa of the smaller operand is shifted by $|e_a - e_b|$ bits to the right, rendering the two exponents equal.
- ADD/SUB. The two resulting mantissas are then added, resp. subtracted, with a standard integer adder.
- ROUND. If the new mantissa has more than p bits, the result is rounded to obtain a number in $\mathbb{F}_p$. The rounding is implemented as a function on the least significant bits of the mantissa.

If $e_a = e_b = e_s = e_r$, the ALIGN module is not needed, since the shift distance is 0. The ROUND module can also be simplified, since the mantissa is not shifted. In this case, the circuit implements *fixed-point* arithmetic: the operation is reduced to the ADD/SUB module. The existence of efficient SAT-encodings for fixed-point arithmetic formulas therefore suggests that reducing the cost of the ALIGN and ROUND modules may improve the performance of a floating-point decision procedure via a SAT-encoding.

Table I shows the number of propositional variables needed for a floating-point adder/subtracter (optimized for propositional SAT, not area or depth), depending on the width p of the mantissa. These numbers confirm that alignment and rounding

Precision	ALIGN	ADD/SUB	ROUND	Total
$p = 5$	295	168	572	1035
$p = 11$	418	252	853	1523
$p = 17$	561	336	1153	2050
$p = 23$	687	420	1447	2554
$p = 29$	813	504	1744	3061
$p = 35$	996	588	2050	3634
$p = 41$	1140	672	2362	4174
$p = 47$	1284	756	2665	4705
$p = 52$	1404	826	2923	5153

TABLE I
NUMBER OF VARIABLES FOR AN FP-ADDER DEPENDING ON p

cause the propositional formula to blow up in size. One way to curb this blow-up is to approximate floating-point operations by reducing the precision p, as we shall do in Section IV.

B. Multiplication and division

A high-level description of a floating-point multiplier/divider is given in Figure 3. Besides the rounder as described above, an FPU implements the following modules for a multiplier/divider:

- ADD/SUB. The exponents of the two operands are first added, for multiplication, resp. subtracted, for division.
- MUL/DIV. The two mantissa are then multiplied, resp. divided.

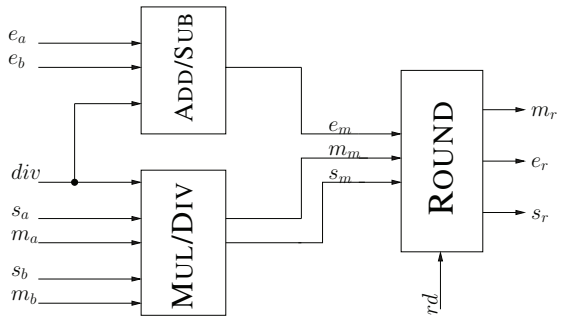


Fig. 3. High-level overview of a floating-point multiplier/divider

Table II shows the number of propositional variables needed for a floating-point multiplier/divider, depending on the width p of the mantissa. As one would expect, the multiplier/divider

Precision	MUL/DIV	ADD/SUB	ROUND	Total
$p = 5$	280	94	674	1048
$p = 11$	982	94	1287	2363
$p = 17$	2188	94	1910	4192
$p = 23$	3898	94	2258	6550
$p = 29$	6112	94	3200	9406
$p = 35$	8830	94	3855	12779
$p = 41$	12052	94	4521	16667
$p = 47$	15778	94	5193	21065
$p = 52$	19268	94	5742	25104

TABLE II
NUMBER OF VARIABLES FOR AN FP-MULTIPLIER DEPENDING ON p

yields a propositional formula that is expensive to decide. One possibility to curb these costs is to reduce the precision p, as this reduces the width of the multiplier. We now discuss how formulas are approximated when p is reduced.

IV. APPROXIMATING FLOATING-POINT ARITHMETIC

Reducing the precision of floating-point operations *approximates* the input formula: a satisfiable formula may become unsatisfiable, or vice versa. In order for the results returned by a SAT solver to still be useful, we need to be aware of the "direction" of the approximation. In this section, we discuss methods for over- and underapproximating floating-point formulas by reducing their precision.

A. Overapproximation

Reducing the precision p of a floating-point operation to p' causes the bits needed for the correct rounding decision to be lost, and the rounding to be based on higher-order bits. It turns out that by making the reduced-precision rounding decision nondeterministic, the reduced-precision formula overapproximates the original.

Definition 3: The *open rounding operation* $\overline{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{F}_p)$ is defined as

$$\overline{rd}_{p,p'}(X) \;:=\; [\,\lfloor X \rfloor_{p'}, \lceil X \rceil_{p'}\,] \cap \mathbb{F}_p\,,$$

where $\lfloor X \rfloor_{p'} := \min_{x \in X} \lfloor x \rfloor_{p'}$ and $\lceil X \rceil_{p'} := \max_{x \in X} \lceil x \rceil_{p'}$. The set $\overline{rd}_{p,p'}(X)$ can be seen as the smallest precision-p floating-point "interval" Y such that for all $x \in X$, the reduced-precision values $\lfloor x \rfloor_{p'}$, $\lceil x \rceil_{p'}$ are in Y. We use the operator $\overline{rd}_{p,p'}$ to define corresponding open floating-point operations.

Definition 4: For an arithmetic operation $\circ : \mathbb{R}^2 \to \mathbb{R}$, the corresponding *open floating-point operation* $\overline{\circledcirc}_{p,p'} : (\mathcal{P}(\mathbb{F}_p))^2 \to \mathcal{P}(\mathbb{F}_p)$ is defined as:

$$X \,\overline{\circledcirc}_{p,p'}\, Y := \overline{rd}_{p,p'}(\{x \circ y \,|\, x \in X, y \in Y\})\,.$$

The new floating-point operation $\overline{\circledcirc}_{p,p'}$ overapproximates the original operation $\circledcirc_p$ in the sense that $\overline{\circledcirc}_{p,p'}$ yields *more* results than $\circledcirc_p$, i.e., $x \circledcirc_p y \in \{x\}\,\overline{\circledcirc}_{p,p'}\,\{y\}$, for any reduced precision $p' \le p$, as the following lemma shows.

Lemma 1: For p, p' with $p' \le p$, $\;x \circledcirc_p y \in \{x\}\,\overline{\circledcirc}_{p,p'}\,\{y\}$.

Proof: By the definitions of $\circledcirc_p$ and rd_p, $x \circledcirc_p y = rd_p(x \circ y) \in \{\lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p\}$. We estimate this set as follows:

$$\begin{array}{lcll} \ldots & \subseteq & [\,\lfloor x \circ y \rfloor_p, \lceil x \circ y \rceil_p\,] \;\cap\; \mathbb{F}_p & \text{[closed interval]} \\ & \subseteq & [\,\lfloor x \circ y \rfloor_{p'}, \lceil x \circ y \rceil_{p'}\,] \;\cap\; \mathbb{F}_p & \text{[nesting prop.]} \\ & = & \overline{rd}_{p,p'}(\{x \circ y\}) & \text{[Def. 3 with } X = \{x \circ y\}] \\ & = & \{x\}\overline{\circledcirc}_{p,p'}\{y\}\,. & \text{[Def. 4]} \end{array}$$

■

One can use the open operations to generate a formula $\overline{\phi}$ that overapproximates the original ϕ. Each floating-point operation $\circledcirc_p$ is replaced by an open version $\overline{\circledcirc}_{p,p'}$, for some reduced precision p'. The reduced precision can be chosen separately for each occurrence of a floating-point operation.

B. Underapproximation

To generate an underapproximation, we devise floating-point operations with *fewer* results than the original operations. Observe that if a floating-point operation with reduced precision p' yields an exact result, then the same result is obtained with the original precision p. Our new floating-point operations are restricted to exact precision p' results. To formalize this idea, we define a modified rounding operator $\underline{rd}_{p,p'}$:

Definition 5: The *no-rounding operator* $\underline{rd}_{p,p'} : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{F}_p)$ is defined as

$$\underline{rd}_{p,p'}(X) \;:=\; X \cap \mathbb{F}_{p'}\,.$$

The quantity $\underline{rd}_{p,p'}(\{x\})$ equals $\{x\}$ if no rounding is required to represent x with precision p', i.e., $x \in \mathbb{F}_{p'}$, and the empty set otherwise, independently of p. Floating-point operations $\underline{\circledcirc}_{p,p'}$ that yield exact results only are defined in analogy to Def. 4, with $\overline{rd}$ replaced by $\underline{rd}$. These operations yield fewer results than their original counterparts $\circledcirc_p$. That is, if $\{z\}$ is the result of the new exact operation $\{x\}\underline{\circledcirc}_{p,p'}\{y\}$, then z is also the result of original operation $x \circledcirc_p y$:

Lemma 2: For p, p' with $p' \le p$, $\;\{x\}\underline{\circledcirc}_{p,p'}\{y\} = \{z\}$ implies $x \circledcirc_p y = z$.

Proof: We have $\{x\}\underline{\circledcirc}_{p,p'}\{y\} = \{x \circ y\} \cap \mathbb{F}_{p'} = \{z\}$, thus $z = x \circ y \in \mathbb{F}_{p'} \subseteq \mathbb{F}_p$. From $x \circ y \in \mathbb{F}_p$, we can conclude $\lfloor x \circ y \rfloor_p = \lceil x \circ y \rceil_p = x \circ y = rd_p(x \circ y) = x \circledcirc_p y = z$. ■

One can use $\underline{\circledcirc}$ to generate an *underapproximation* of a formula ϕ, by replacing each floating-point operation with a version that is exact for some reduced precision p'.

In case the constructed underapproximation is shown to be unsatisfiable, nothing can be concluded on the satisfiability of ϕ. One way to resolve the dichotomy between over- and underapproximations is to integrate *both* into an abstraction-refinement framework.

V. The Mixed Abstraction Framework

A. Overview

In Section IV, we have presented over- and underapproximation techniques to simplify a given floating-point formula ϕ. Many existing procedures build *either* over- or underapproximations, depending on whether the goal is to show satisfiability or unsatisfiability. The two types of approximation guarantee a definite decision on the satisfiability of ϕ only in cases that are *orthogonal* for the two types. We therefore propose to combine them in a concerted effort towards analyzing ϕ.

To this end, we propose the abstraction framework shown in Figure 4, which checks the satisfiability of the input formula ϕ. We first identify a set of eligible *transformations*. A transformation is a mapping that turns a FPA formula β into a new one that over- or underapproximates β, for example by replacing some floating-point operation by its open version, as suggested in Section IV. The set of transformations is accordingly partitioned into subsets $Over$ and $Under$. At the beginning of the loop indicated in the figure, an implementation selects some of the eligible transformations and applies them to ϕ in a particular order. Note that the resulting formula ψ in general neither over- nor underapproximates ϕ (hence called "mixed").

Exit points of the loop. Formula ψ is then subject to a satisfiability check. Depending on the outcome of this check, the loop can be exited — with a definite answer — if:

(i) ψ is satisfiable, and the assignment α returned by the solver (suitably extended) satisfies ϕ as well. In this case, the overall answer is "SAT". Or:

(ii) ψ is unsatisfiable, and the resolution proof P returned by the solver is valid for ϕ, too. In this case, the overall answer is "UNSAT".

Refinement. If neither case (i) or (ii) applies, the approximation needs to be refined. This is done by removing some transformations from $Over$ if ψ was found to be spuriously satisfiable, otherwise from $Under$. Which transformations to

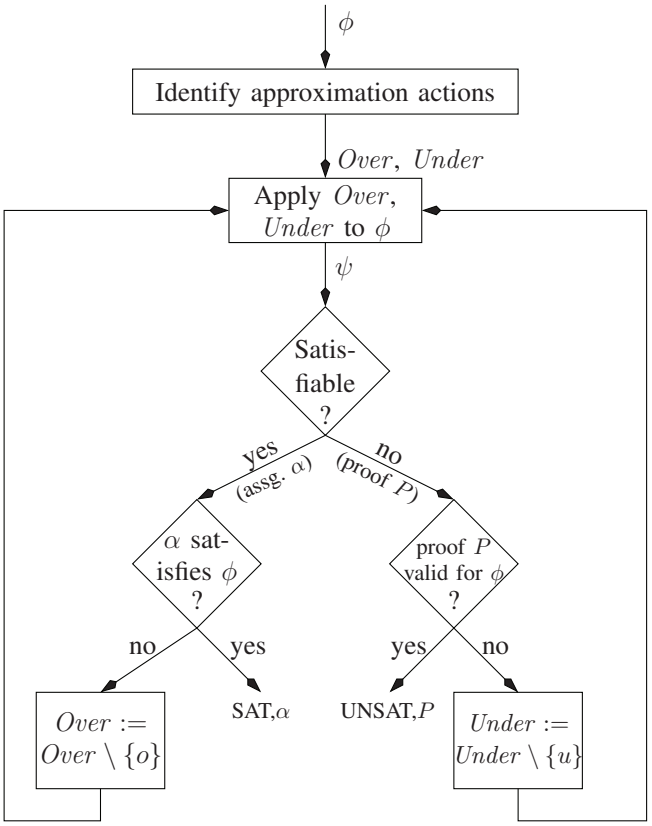


Fig. 4. The Framework of Mixed Abstraction

select for removal is an important implementation decision, which we discuss below in Section VI-B. The loop in Figure 4 is then reentered, and some transformations from the new sets $Over$ and $Under$ are applied to ϕ.

Notice that the procedure can be implemented in a both incremental and backtrackable fashion, provided the underlying SAT solver is incremental and backtrackable.

Property 1: Given a formula ϕ, any algorithm that implements the framework in Figure 4, starting with finite sets $Over$ and $Under$ of transformations, terminates and returns "SAT" if ϕ is satisfiable, "UNSAT" otherwise.

Proof:

(a) Partial correctness: The algorithm outputs "SAT" only in the case that the assignment α was validated successfully against the original formula ϕ. It outputs "UNSAT" only in the case that an unsatisfiability proof for ψ was found to be a valid proof of unsatisfiability for ϕ.

(b) Termination: In each round in which the algorithm does not exit, at least one element is removed from $Over$ or from $Under$. When both sets are exhausted, ψ and ϕ are equivalent, and one of the two exit conditions is trivially satisfied. ■

Note that the correctness property is independent of the distribution and scheduling of over- and underapproximations, and of the strategy for selecting elements o or u to remove in each iteration. Furthermore, it can be shown that finiteness of the sets $Over$ and $Under$ is not even required if one defines a suitable well-quasi-order on the set of all approximation transformations.

The Mixed Abstraction framework generalizes several other abstraction-refinement approaches to satisfiability-checking a formula. The classical CEGAR paradigm *conservatively abstract–check–refine* can be seen as an instance of Figure 4 where the set $Under$ is empty. Therefore, the test whether an unsatisfiability proof for ψ is valid for ϕ can be skipped in favor of an immediate answer "UNSAT". The method presented in [4] has the property that the sequence of approximations obtained from ϕ strictly alternates between strict over- and strict underapproximations. As we experimentally compare this alternating scheme to our own implementation of mixed abstraction, we sketch first how the method of [4] can be applied in a floating-point environment.

VI. Implementing Mixed Abstractions

A. Alternating Abstractions for Floating-Point Arithmetic

The alternating approach by Bryant et al. [4] to integrating over- and underapproximations was implemented for integer bit-vector arithmetic formulas. If the SAT-check on an approximated formula is inconclusive, information obtained from the SAT checker is used to generate a refined approximation of the opposite type, and the procedure repeats.

We can use the approximation methods presented in Section IV to apply the alternating approach to floating-point arithmetic, as follows. We begin with an overapproximation $\overline{\phi}$ of ϕ. To obtain $\overline{\phi}$, the transformations in $Over$ replace all floating-point operations $\circledcirc$ by $\overline{\circledcirc}_{p,p'}$, for some initial reduced precision p'. Since $Under$ is not applied, $\overline{\phi}$ is an overapproximation.

If $\overline{\phi}$ is unsatisfiable the procedure terminates. Otherwise, the decision procedure yields an assignment α. If α also satisfies ϕ, the procedure halts and returns α as a witness. If α is spurious, we extract from it the operands a, b, and the result r of each occurrence of $\overline{\circledcirc}_{p,p'}$. In case $r \neq \circledcirc_p$ we conclude that r is a spurious result and we refine (increase) the precision of $\overline{\circledcirc}_{p'}$; otherwise p' is left unchanged.

Next, the decision procedure builds a refined underapproximation $\underline{\phi}$ as explained in Section IV-B. In this iteration, the transformations in $Under$ replace all occurrences of $\circledcirc$ by $\underline{\circledcirc}_{p,p'}$ for the refined precisions p'; no transformation from $Over$ is applied. In case $\underline{\phi}$ is satisfiable, the procedure terminates and returns α as an assignment for ϕ. Otherwise, the decision procedure yields an UNSAT proof P for $\underline{\phi}$. If P is also a proof for ϕ, the procedure halts and returns P. Otherwise, it checks whether the constraint $X \cap \mathbb{F}_p$ of an exact operation $\underline{\circledcirc}_{p'}$ (Definition 5) is contained in P. If it is, the precision is increased; otherwise it is left unchanged. The next iteration constructs a refined overapproximation. Altogether, this yields a sound and complete decision procedure alternating between over- and underapproximations.

The problem with this approach is that the alternating schedule of over- and underapproximations often leads to ineffective approximations, as some formulas are not amenable to effective overapproximations, while others do not permit

effective underapproximations. As an example, consider the non-associativity formula $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$. This formula is satisfiable, as floating-point addition $\oplus$ is not associative. Satisfiability cannot be proved using an overapproximation. On the other hand, every strict underapproximation of this formula turns out to be unsatisfiable. Thus, this formula cannot be decided using **either** strict over- **or** strict underapproximations.

Our experimental results (Section VII) confirm these predictions for realistic formulas. The lesson is that an implementation of Mixed Abstraction should not be "forced" to apply either type of abstraction for pure schedule reasons. Instead, the structure of the formula itself should dictate how to approximate it. This leads to our approach of "genuinely mixed abstractions", which we present in the following.

B. Genuinely Mixed Abstractions

We now detail our implementation of the mixed abstraction framework. The selection of abstractions is determined by the structure of the formula, and which approximations are most effective on it. We begin with both a very coarse overapproximation and a strong underapproximation: the result of the operation is completely nondeterministic and forced to zero at the same time. Depending on the outcome of the satisfiability check of ψ, either one of these approximations is refined, gradually lifting constraints or gradually increasing the precision of the operator.

The simulation of α on ϕ in the left branch in Figure 4 can be preceded by a check whether any transformation in $Over$ was applied to ϕ. If not, ψ is guaranteed to be an underapproximation of ϕ, and "SAT" and α are returned immediately. Otherwise, α is suitably extended to an assignment for ϕ and checked for satisfaction of ϕ, as suggested by the figure.

A simple and efficient pre-check whether an unsatisfiability proof P for ψ is extendable to ϕ can be performed by computing the set $Var(P) \cap Var(Under)$. This set contains the variables occurring in (the clauses of) the proof P that are involved in any of the underapproximation transformations in $Under$. If empty, we conclude that the underapproximation transformations applied to ϕ are not responsible for the unsatisfiability result for ψ, which hence applies to ϕ, too. The emptiness test can obviously be optimized by checking first whether any transformation in $Under$ was applied to ϕ.

If none of the exit tests succeeds, the algorithm selects approximation transformations o or u to remove at the end of the loop body, in order to refine the approximation. Let us look first at the case that ψ was found to be satisfiable, but the assignment α is spurious (does not satisfy ϕ). If there exists a transformation $o \in Over$ such that α does not satisfy the formula obtained by simplifying ϕ using all transformations *except* o, we can select such an o: its removal guarantees that the spurious assignment α disappears in the next round. If there is no such transformation, we select some o that affects the variables occurring in the spurious assignment α.

If ψ was found to be unsatisfiable, the algorithm computes the set $Var(P) \cap Var(Under)$, as discussed above. If this set is non-empty, then there exists at least one $u \in Under$ such that $Var(u) \cap Var(Under)$ is non-empty. The algorithm picks such an element u for removal from $Under$.

After refining the sets $Over$ and $Under$ as described, Figure 4 suggests to apply the refined (and smaller) sequence of approximations to ϕ again. In practice, approximations selected for removal in the previous step are revoked, so that only local modifications to ψ are necessary. This allows the subsequent satisfiability check to be done incrementally, without restarts of the SAT solver.

VII. Experimental Results

A. Implementation and Benchmarks

We have implemented the algorithm proposed in this paper in combination with a standard bit-flattening decision procedure for integer bit-vector arithmetic. The procedure supports all operators required to model ANSI-C programs. The procedure is fully incremental: the clauses for those parts of the encoding that are not modified, and any clauses learned from those, are retained between iterations. We use MiniSAT2 [6] as our SAT solver. The performance of the integer bit-vector decision procedure we use is comparable to that of a state-of-the-art SMT-$\mathcal{BV}$ solver.

The benchmarks we use are derived from a variety of publicly-available C programs, selected from the SNU real-time [7] and the Mediabench benchmarks [8]; the programs we have selected make extensive use of single- or double-precision floating-point arithmetic. Our encoding is able to support changes of the rounding mode during the program execution, as the rounding mode may be set separately for each individual operator in the formula. However, none of the benchmark programs makes use of this feature, and we have used "round to nearest even" uniformly. We have not observed a significant impact of the specific rounding mode on the performance of either the full encoding or the abstraction-refinement procedure.

In order to obtain verification conditions, we have manually annotated our benchmarks with properties in the form of arithmetic assertions; a few of the programs already contain some (light-weight) assertions. The main property we check is the possibility of arithmetic exceptions, as defined in the IEEE floating-point arithmetic standard. These properties turn out to be difficult; an instance of a counterexample is arithmetic underflowing to zero and a subsequent division of two such numbers, which results in NaN. Such counterexamples can only be obtained if encoding artifacts such as denormal numbers are modelled in a precise manner.

After the annotation, we pass the programs to CBMC [9], which generates a total of 119 FPA decision problems. A satisfying assignment for such a formula corresponds to a counterexample that demonstrates that an assertion can be violated. Our experiments were performed on a machine running Linux on an Intel Xeon CPU with 3 GHz and 4 MB cache.

B. Results

Figure 5 compares the runtime of the mixed abstraction with the full flattening, for a timeout of 1000 s and a memory

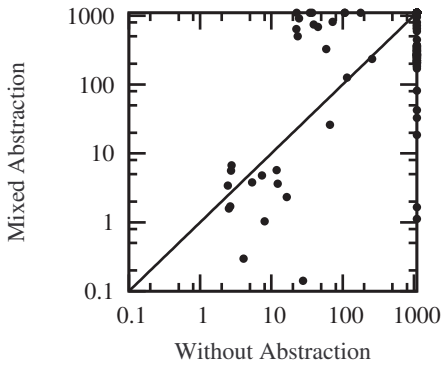


Fig. 5. Runtime comparison of full flattening and the mixed abstraction

limit of 2 GB. The mixed abstraction is not dominating; in some cases (6 out of 119), the final abstraction contains almost all operators at full precision, and as a result, the immediate full flattening is faster. On the other hand, there are 58 large instances for which the abstraction-refinement procedure terminates within the timeout, whereas the full flattening aborts due to excessive memory consumption.

Let us look at some benchmarks in more detail. For a timeout of 7 hours, Table III provides a comparison of the performance of three methods: the full flattening, an alternating abstraction as proposed in [4], and the mixed abstraction. The entries illustrate particular weaknesses and strengths of the three procedures. First of all, even small programs may result in decision problems that are simply too large (or too hard) for a full flattening. Similarly, there are instances in which the alternating abstraction is unable to proceed beyond the first iteration, as either the pure under- or overapproximation is already too hard. The mixed abstraction terminates on a number of instances that are too hard for the other procedures, but typically requires a large number of iterations. This is owed to the extremely coarse initial abstraction, combining both over- and underapproximations. As long as the intermediate abstractions are solvable, the alternating abstraction may converge quicker to a solution (e.g., consider sqrt.c, claim 2).

VIII. Related work

A popular approach to verifying software with floating-point operations is to use *proof assistants*, i.e., programs that prove theorems under the guidance of a human expert. A variety of assistants have been used to prove upper bounds for the deviation from the *real* arithmetic result of a calculation. Examples include proofs using HOL [10], HOL-Light [11]–[13] and ACL2 [14], [15]. However, in case no proof is found, proof assistants typically return little evidence as to whether the proof goal was actually invalid, or whether the proof strategy was too weak to establish its validity.

An alternative approach is to use *abstract interpretation* [2] together with domains that can soundly approximate floating-point computations for classical static analysis. The static analyzer ASTRÉE [16] uses a number of domains which can soundly abstract floating-point computations, including intervals, octagons, polyhedra, and ellipsoid domains.

The adaptation of abstract domains for floating-point numbers is a non-trivial problem due to issues of rounding, the possibility of overflows and underflows, and division by zero errors. Relational abstract domains such as the octagon domain rely on associativity and distributivity of arithmetic operations. These properties do not hold for floating-point numbers. In ASTRÉE, floating-point expressions are therefore approximated by linear expressions over the real field with interval coefficients [17] before they are transformed into their target abstract domains. In [18], a floating-point polyhedra abstract domain based on these ideas is presented. Their linearisation technique is implemented in the APRON library for static analysis [19], which provides sound handling of floating-point expressions for a number of abstract domains.

The propagation of floating-point rounding errors has also been studied extensively in the framework of abstract interpretation [20]–[23]. Such analyses allow to quantify deviations of floating-point computations from their exact result in arithmetic over the reals. Verifying floating-point programs using abstract interpretation shares with our method the advantage of being fully automatic. However, in case the property does not hold, it is usually difficult to obtain a counterexample that can be reproduced on the actual program.

Neither interactive theorem provers nor the use of abstract domains enjoy the characteristics of a *decision procedure*, namely completeness and full automation in deciding floating-point expressions. There has been some work on decision procedures for floating-point arithmetic in the field of *constraint satisfaction programming (CSP)*. Solvers for CSP instances containing floating-point constraints have been applied to automated test-vector generation [24], [25]. This approach combines filtering the possible values of variables using interval techniques with a search procedure for finding actual floating-point values inside these intervals. Their algorithm is mainly geared towards test-vector generation, not verification. The authors state that in some cases where the calculated intervals overapproximate the concrete variable values too coarsely, their approach is unable to terminate with an answer in reasonable time. This includes the case where no such concrete solutions exist.

In this paper, we are interested in verification of software using floating-computations. A different, but related field of research is the verification of floating-point hardware. The work described in [26] is an example of such an approach and provides an overview over the relevant literature.

IX. Conclusion

We have presented an algorithm for iteratively approximating a complex formula by mixing both under- and over-approximations to obtain a formula ψ. In contrast to prior work, ψ need not be an over- nor an underapproximation and can therefore be constructed in a way that yields formulas that are easy to solve. Experimental results indicate improved

Benchmark	Lines of Code	Satisfiable?	No Abstr.	Alternating [4]		Mixed	
			time (s)	time (s)	#iter.	time (s)	#iter.
qurt.c, claim 1	109	no	25	15	1	2	15
qurt.c, claim 2	109	no	25	15	1	0.6	7
qurt.c, claim 3	109	no	25	15	1	1	13
qurt.c, claim 4	109	no	OM	OM	1	478	103
qurt.c, claim 5	109	no	25	15	1	1.2	15
qurt.c, claim 6	109	no	25	15	1	0.6	7
qurt.c, claim 7	109	no	6716	OM	1	84	86
sqrt.c, claim 1	51	no	24	TO	3	13589	44
sqrt.c, claim 2	51	yes	9	608	2	TO	107
minver.c, claim 1	156	no	1	1	1	0.1	1
minver.c, claim 2	156	yes	2	2	2	0.1	1
sin.c, claim 1	46	no	13864	1892	1	281	47
sin.c, claim 2	46	no	13831	1894	1	281	47
sin.c, claim 3	46	no	TO	TO	3	1074	63
gaussian.c, claim 1	108	no	TO	TO	1	14437	137

TABLE III
COMPARISON OF FULL FLATTENING TO ALTERNATING AND MIXED ABSTRACTION

robustness compared to a plain flattening and to an abstraction-refinement scheme based on an alternation of over- and underapproximations.

The algorithm supports incremental solving, is complete, and produces witnesses. In particular, the ability to generate counterexamples for invalid specifications is essential not only for debugging, but also for the high-impact field of *automated test-vector generation*, where counterexamples to carefully crafted specifications translate into test cases meeting certain coverage criteria.

One aspect of future work is how to vary the exponent width r in order to approximate a floating-point operation. While the operations on the exponent contribute only the smaller part of the propositional encoding, many programs exist that only exercise a very small range of exponent values.

REFERENCES

[1] Gander, W.: Heisenberg effects in computer-arithmetic (2005) http://www.inf.ethz.ch/news/focus/res_focus/april_2005.

[2] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, New York, USA, ACM (1977) 238–252

[3] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. (2000) 154–169

[4] Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: Deciding bit-vector arithmetic with abstraction. In: TACAS. (2007) 358–372

[5] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149

[6] Eén, N., Sörensson, N.: An extensible sat-solver. In: SAT. Number 2919 in LNCS, Springer (2003) 502–518

[7] SNU Computer Architecture and Network Laboratory: (SNU Real-Time Benchmarks) http://archi.snu.ac.kr/realtime/benchmark.

[8] Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In: MICRO. (1997) 330–335

[9] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. (2004) 168–176

[10] Carreno, V.A.: Interpretation of IEEE-854 floating-point standard and definition in the HOL system. Technical report, NASA Langley Research Center (1995)

[11] Harrison, J.: Formal verification of square root algorithms. Form. Methods Syst. Des. **22** (2003) 143–153

[12] Harrison, J.: Formal verification of floating point trigonometric functions. In: FMCAD. (2000) 217–233

[13] Harrison, J.: Floating point verification in HOL light: The exponential function. In: AMAST. (1997) 246–260

[14] Russinoff, D.M.: A mechanically checked proof of correctness of the AMD K5 floating point square root microcode. Form. Methods Syst. Des. **14** (1999) 75–125

[15] Russinoff, D.M.: A case study in fomal verification of register-transfer logic with ACL2: The floating point adder of the AMD AthlonTM processor. In: FMCAD '00: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, London, UK, Springer-Verlag (2000) 3–36

[16] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI, ACM (2003) 196–207

[17] Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: ESOP. Volume 2986 of Lecture Notes in Computer Science., Springer (2004) 3–17

[18] Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: APLAS. Volume 5356 of Lecture Notes in Computer Science., Springer (2008) 3–18

[19] Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: CAV. (2009, to appear)

[20] Goubault, E.: Static analyses of the precision of floating-point operations. In: SAS. (2001) 234–259

[21] Goubault, E., Martel, M., Putot, S.: Asserting the precision of floating-point computations: A simple abstract interpreter. In: ESOP. Volume 2305 of Lecture Notes in Computer Science., Springer (2002) 209–212

[22] Martel, M.: Static analysis of the numerical stability of loops. In: SAS. Volume 2477 of Lecture Notes in Computer Science., Springer (2002) 133–150

[23] Putot, S., Goubault, E., Martel, M.: Static analysis-based validation of floating-point computations. In: Numerical Software with Result Verification. Volume 2991 of Lecture Notes in Computer Science., Springer (2003) 306–313

[24] Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: CP. Volume 2239 of Lecture Notes in Computer Science., Springer (2001) 524–538

[25] Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Softw. Test., Verif. Reliab. **16** (2006) 97–121

[26] Jacobi, C., Berg, C.: Formal verification of the VAMP floating point unit. Formal Methods in System Design **26** (2005) 227–266

[27] Monniaux, D.: On using floating-point computations to help an exact linear arithmetic decision procedure. In: CAV. (2009, to appear)

Safety First: A Two-Stage Algorithm for LTL Games

Saqib Sohail and Fabio Somenzi
University of Colorado at Boulder

***Abstract*—In the game theoretic approach to the synthesis of reactive systems, specifications are often given as a conjunction of linear time properties. An implementation can be obtained from a winning strategy derived from a suitable generalized parity game in which each property produces a parity acceptance condition. Safety and persistence properties usually make up the majority of the specification. We show how this can be exploited to play the game in two stages and substantially speed up synthesis without sacrificing generality and conciseness of specification.**

I. Introduction

The game-theoretic approach to the synthesis of reactive systems is the focus of renewed attention thanks to the significant algorithmic advances of the last few years. While the doubly exponential bound established in Pnueli and Rosner's seminal paper [1] suggests that challenges to scalability will persist, there is increasing hope that synthesis algorithms may be applied to the design and diagnosis of intricate, safety-critical protocols.

In the game-theoretic approach to synthesis, the specification of a reactive system is converted to a two-player game. One player represents the environment to which the system must react; the other player represents the system itself. The game is a finite graph and a play is an infinite sequence of vertices. With this setup, a winning strategy for the system player—if it exists—yields an implementation of the specification.

A system's intended behavior is often described by several simple properties, each given as either a formula in linear temporal logic (LTL) or as an ω-regular automaton. In a naive approach, all formulae and automata are reduced to one deterministic automaton, whose transition structure provides the game graph and whose acceptance condition is taken as the winning condition. This approach suffers from the high cost of determinization [2], which is prohibitive for even moderate-sized automata.

Several remedies have been proposed, ranging from restricting specifications to a subset of ω-regular properties that allows one to use algorithms more efficient than the general ones [3], to avoiding determinization through alternate constructions [4], [5], [6]. In [7] we have presented an approach based on deriving a deterministic parity automaton for each property in the specification, thus avoiding in most cases the application of the expensive determinization procedure to large automata. The price to be paid is the switch from the solution of a parity game [8] to that of a more complex conjunctive generalized parity game—one equipped with multiple parity winning conditions that must be simultaneously satisfied [9]. In [7] we argued that this trade-off is advantageous because the most expensive operation—namely, the solution of the game—can be done symbolically, that is, by an algorithm that manipulates characteristic functions of sets rather than their members. (Determinization, on the other hand, is carried out by an explicit enumeration algorithm.)

Since the number of components of a conjunctive generalized parity winning condition sharply affects the time required to solve the game, one may conjoin some properties before translating them to deterministic automata as long as no blow up occurs. The choice of the properties to be merged is, however, heuristic. In contrast, in this paper we propose a two-stage approach to improve the solution of the generalized parity game. We prove that safety and persistence properties can be dealt with before the rest of the properties without affecting the algorithm's completeness. Safety and persistence properties make up the bulk of most specifications of reactive system. Their presence allows us to improve the worst-case runtime of the game solving algorithm and leads to significant time savings.

The rest of this paper is organized as follows. Section II recalls the notions on ω automata, temporal logic, and games that are relevant to this paper. Sections III and IV describe the two-stage algorithm and our implementation. Section V discusses related work. Section VI presents our experimental results and Sect. VII concludes.

II. Automata, Logic, and Games

A finite automaton on ω-words $\langle \Sigma, Q, q_{\text{in}}, \delta, \alpha \rangle$ is defined by a finite alphabet Σ, a finite set of states Q, an initial state $q_{\text{in}} \in Q$, a transition function $\delta : Q \times \Sigma \to 2^Q$ that maps a state and an input letter to a set of possible successors, and an acceptance condition α that describes a subset of Q^ω, that is, a set of infinite sequences of states. A *deterministic* automaton is such that $\delta(q, \sigma)$ is a singleton for all states $q \in Q$ and all letters $\sigma \in \Sigma$. A run of automaton M on ω-word $w = w_0 w_1 \ldots$ is a sequence $q_0, q_1, \ldots$ such that $q_0 = q_{\text{in}}$, and for $i \geq 0$, $q_{i+1} \in \delta(q_i, w_i)$. A run is accepting iff (if and only if) it belongs to the set described by α, and a word is accepted iff it has an accepting run in M. The subset of Σ^ω accepted by M is the (ω-regular) language of M.

Several types of acceptance conditions α are in use. Here we are concerned with Büchi [10], and parity [11], [12] acceptance conditions. Both conditions are about the set of states $\mathsf{inf}(\rho)$ that occur infinitely often in a run ρ. A run ρ is accepting for a Büchi acceptance condition $F \subseteq Q$ iff $\mathsf{inf}(\rho) \cap F \neq \emptyset$. A parity acceptance condition is specified by assigning a *priority* to each state of the automaton. Letting $[k] = \{i \mid 0 \leq i < k\}$, a parity condition of index k is a function $\pi : Q \to [k]$. A run ρ is accepting iff

This work was supported in part by SRC contract 2008-TJ-1365.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

$\max\{\pi(q) \mid q \in \mathsf{inf}(\rho)\}$ is odd; that is, iff the highest recurring priority is odd.

Büchi and parity acceptance conditions may be *generalized*. A generalized Büchi condition consists of a collection $\mathcal{F} \subseteq 2^Q$ of Büchi conditions. A run ρ is accepting for a generalized Büchi condition iff it is accepting according to each $F \in \mathcal{F}$. A generalized parity condition may be either conjunctive or disjunctive and is given as a collection Π of priority functions. A run ρ is accepting according to a conjunctive (disjunctive) condition Π iff it is accepting according to each (some) $\pi \in \Pi$. Disjunctive and conjunctive generalized parity conditions are dual: if one of the two players has a winning condition of one type, the opponent has a winning condition of the other type.

An ω-regular automaton equipped with a Büchi acceptance condition is called a Büchi automaton; likewise for the other acceptance conditions. In this paper, we adopt popular three-letter abbreviations to designate different types of automata. The first letter of each abbreviation distinguishes nondeterministic (N) from deterministic (D) structures. The second letter denotes the type of acceptance condition: Büchi (B) or parity (P). The final letter indicates that the automata read infinite words (W) or infinite trees (T). As examples, NBW designates a nondeterministic Büchi automaton (on infinite words), while DPW is the acronym for a deterministic parity automaton (also on infinite words).

A subset $S \subseteq \Sigma^\omega$ is a *safety property* iff every word not in S has a prefix that cannot be extended to a word in S. A *liveness property* $L \subseteq \Sigma^\omega$ is such that every finite word in Σ^* can be extended to a word in L. Safety properties correspond to closed sets of the product topology of Σ^ω, while liveness properties correspond to dense sets [13].

DBWs are less expressive than NBWs; accordingly, determinization is only possible in general by switching to a more powerful acceptance condition. Piterman [2] has introduced a procedure to produce a DPW from an NBW. The construction extends the well-known subset construction for automata on finite words. Rather than labeling each state of the deterministic automaton with a subset of states of the NBW, it labels it with a tree of subsets. As a result, the upper bound on the number of states of the DPW derived from an NBW with n states is n^{2n+2}. This fast-growing function discourages determinization of large NBWs.

Linear Time Logic (LTL) [14], [15] is a popular temporal logic for the specification of nonterminating reactive systems. LTL formulae are built from a set of atomic propositions, Boolean connectives, and basic temporal operators X (next), U (until), and R (releases). Derived operators G (always) and F (eventually) are usually included for convenience. Procedures exist (e.g., [16]) to translate an LTL formula into an NBW that accepts the language defined by the formula. If, on the one hand, not all ω-regular languages can be expressed in LTL, DBWs are not sufficient to translate all of LTL. Sistla [17] has shown that LTL formulae in negation normal form that do not use the until operator define safety properties.

Deterministic ω-regular automata can be used to define infinite games [18] in several ways. Here we consider turn-based and input-based two-player games, in which Player 0 (the *antagonist*) and Player 1 (the *protagonist*) move a token along the transitions of the automaton. If the resulting infinite sequence of states is accepted by the automaton, Player 1 wins, otherwise Player 0 wins. In *turn-based* games the set of states Q is partitioned into Q_0 (antagonist states) and Q_1 (protagonist states). Each player moves the token from its states by choosing a letter from Σ and the corresponding successor according to δ. In *input-based* games, the alphabet Σ is the Cartesian product $\Sigma_0 \times \Sigma_1$ of an antagonist alphabet and a protagonist alphabet. In state q, Player i chooses a letter $\sigma_i \in \Sigma_i$. The token is then moved to $\delta(q, (\sigma_0, \sigma_1))$. Turn-based games are games of perfect information, whereas in input-based games a player may have full, partial, or no advance knowledge of the other player's choices. The amount of information available to one player obviously affects its ability to win the game. In particular, if Player 1 has no advance knowledge of the opponent's moves, the synthesized reactive system is constrained to be of Moore type. Since this may be restrictive, in our input-based games we assume that Player 1 has partial advance knowledge of the other player's choices. First Player 0 chooses a subset of Σ_0 and discloses it to Player 1; then Player 1 chooses a letter from Σ_1 and discloses it to Player 0; finally Player 0 selects a letter from the chosen subset. Therefore, our input-based games are reduced to turn-based games, and we only discuss the latter form. The three-phase selection process allows us to synthesize Mealy controllers whose outputs may depend on some inputs, but not on others.

The existence and computation of winning strategies are central problems in the study of infinite games. A strategy is a function that defines which letter a player should choose at each move. A strategy for Player i in a turn-based game can be defined equivalently as either a function $\tau_i : Q^* \times Q_i \to \Sigma$, or as a function: $\tau_i : Q_i \times S_i \to \Sigma \times S_i$. The set S_i is the player's *memory*; according to its cardinality, strategies are classified as *infinite memory*, *finite memory*, and *memoryless* (or *positional*). A strategy τ_i is *winning* for Player i from a given state of the automaton iff victory is secured from that state regardless of the opponent's choices as long as Player i plays according to τ_i.

The acceptance condition of the automaton translates into the *winning condition* of the game. We consider Büchi and parity games, and their generalized counterparts. All these games are *determinate* [19]; that is, from each state of the automaton if a player has no winning strategy then the opponent has one. Büchi, parity, and disjunctive generalized parity games admit memoryless strategies. The others—generalized Büchi, and conjunctive generalized parity games—admit finite memory strategies [20].

If the winning condition of an infinite game is given as an LTL formula on the states of the automaton an *LTL game* is obtained. Such a game can be solved by translating the formula into a deterministic ω-automaton and composing it with the graph of the given automaton. Not all LTL formulae have an equivalent DBW. Therefore, determinization to a more powerful type of automaton is required in general. With Piterman's improvement of Safra's construction [2], the parity condition is the natural choice.

In [7] we have presented an algorithm that takes as input

a collection of LTL formulae and NBWs over an alphabet $\Sigma = \Sigma_0 \times \Sigma_1$. The input is converted into a conjunctive generalized parity game with one parity function for each formula and automaton. At each turn, Player 0 chooses a letter from Σ_0 and Player 1 chooses a letter from Σ_1. The objective of Player 1 is to satisfy the conjunctive generalized parity acceptance condition. A winning strategy for Player 1 from the initial state of the parity automaton thus corresponds to an implementation of a reactive system that reads inputs from alphabet Σ_0 and produces outputs from alphabet Σ_1. The reactive system satisfies all the linear-time properties given as LTL formulae or Büchi automata from its initial state. If no such winning strategy exists, there exists no implementation of the given specification. The "classical" algorithm described in [9] is used to compute winning strategies for generalized parity conditions. This algorithm is based on [21], which in turn extends Zielonka's algorithm for parity conditions [20].

Zielonka's algorithm is representative of a class of procedures used to compute winning strategies of a parity game. Though other algorithms have better worst-case complexity, it has the advantage of lending itself to efficient symbolic implementation. Zielonka's algorithm, when applied to the solution of Büchi and co-Büchi games (regarded as two-color parity games), is as efficient as dedicated (symbolic) algorithms. It is simpler—and therefore easier to present succinctly—than the procedure of [21] or the "classical" procedure of [9]; however, it is close enough to those algorithms that its discussion sheds light on them as well.

The pseudocode shown in Fig. 1 extends the one given in [22] with the computation of the strategies. The reader is referred to [22] for a detailed explanation of its operation. Here we focus on strategy computation, which is related to our approach. The algorithm of Fig. 1 takes as input a game graph G, whose set of vertices Q is partitioned into Q_0 and Q_1, and a parity condition; it returns the winning positions of each player and two sets of transitions T_0 and T_1 of the game graph G such that the transitions in T_i belong to winning strategies of Player i. Specifically, T_i contains at least a transition from each vertex in $U_i \cap Q_i$, and any strategy for Player i that is consistent with T_i is a winning strategy from all winning positions of Player i. A strategy σ is consistent with T_i iff $\forall u \in (U_i \cap Q_i) . (u, \sigma(u)) \in T_i$.

A key step in the algorithm of Fig. 1 is the computation of the i-attraction in G of a set of vertices A, that is, the set of vertices $\mathrm{attr}_i(G, A)$ of G from which Player i can force a visit of some vertex in A. The computation of $\mathrm{attr}_i(G, A)$ is a least fixpoint computation, in which $\mathrm{attr}_i(G, A)$ is initialized to A and the set of attraction transitions $\Delta_i(G, A)$ is initially empty. Vertices of G in Q_i (controlled by Player i) are added if they have at least one transition into a vertex already acquired to $\mathrm{attr}_i(G, A)$, while vertices of G in Q_j are added if all their transitions lead to vertices already acquired to $\mathrm{attr}_i(G, A)$. When a vertex in Q_i is added to $\mathrm{attr}_i(G, A)$, the transitions from it to vertices already in $\mathrm{attr}_i(G, A)$ are added to $\Delta_i(G, A)$.

If U_j is empty at Line 6 of the algorithm, then Player i wins from all vertices of the current G whenever the play stays in G. In this case, T_i is obtained as the union of three

```
1   algorithm ZIELONKA(G)
2   if G = (∅, ∅) then return (∅, ∅, ∅, ∅)
3   d ← d(G); A ← A_d(G)
4   i ← d mod 2; j ← ¬i
5   (U_0, U_1, T_0, T_1) ← ZIELONKA(G \ attr_i(G, A))
6   if (U_j = ∅) then
7       U_i ← V(G)
8       T_i ← Δ_i(G, A) ∪ T_i ∪ {(u, v) | u ∈ A ∩ Q_i, v ∈ U_i}
9   else
10      (U_0, U_1, W_0, W_1) ← ZIELONKA(G \ attr_j(G, U_j))
11      U_j ← V(G) \ U_i
12      T_i = W_i
13      T_j = T_j ∪ Δ_j(G, U_j) ∪ W_j
14  fi
15  return (U_0, U_1, T_0, T_1)
```

Fig. 1. Zielonka's algorithm for parity games. In the code, $d(G)$ is the largest color of G and $A_d(G)$ is the set of vertices of G with color d

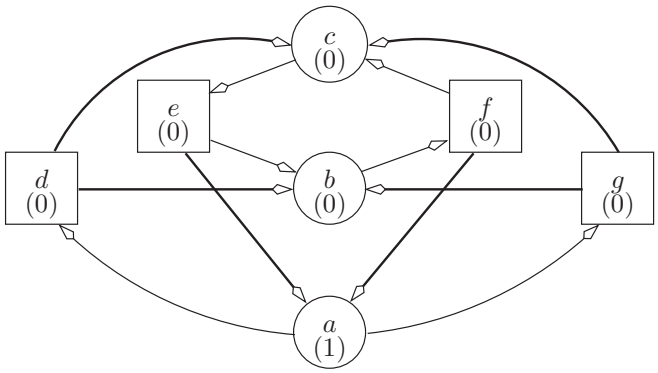


Fig. 2. A parity game. Vertex colors are shown in parentheses

sets: The set $\Delta_i(G, A)$ produced by the attraction computation at Line 5, the set of transitions for Player i returned by the recursive call at Line 5, and the transitions from vertices in $A \cap Q_i$ to vertices in G.

If U_j is not empty at Line 6, then Player j wins from every vertex in U_j using the transitions computed in the recursive call at Line 5. Moreover, it wins from every vertex in $\mathrm{attr}_i(G, A)$ from which it can force a visit of U_j. To force such a visit, it uses the transitions in $\Delta_j(G, U_j)$. (Note that if any vertex is added to $\mathrm{attr}_j(G, U_j)$ by the attraction computation, at least one such vertex is from A.) The second recursive call (Line 10) returns winning positions and transitions for the residual subgraph. For Player i these are the only winning positions and transitions in G. For Player j, the resulting positions and transitions are added to those computed at Lines 5 and 10.

Figure 2 shows a parity game with two colors. Boxes denote vertices in Q_1 and circles denote vertices in Q_0. Zielonka's algorithm returns all vertices of the graph in U_1. The bold arrows denote transitions in T_1. Since both transitions out of vertices d and g are in T_1, there are four distinct memoryless winning strategies for Player 1.

Note that transitions (e, b) and (f, c) are not in T_1. A strategy that used both of them would have to use memory to prevent the play from cycling through b, f, c, and e without

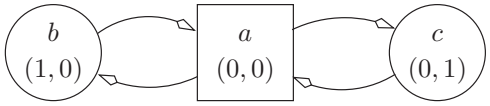


Fig. 3. A simple generalized parity game

visiting a. Exclusion of (e, b) and (f, c) from T_1 prevents such cycles, but also rules out eight memoryless strategies that include one but not the other. The computation of attractions imposes a preorder on the vertices in the fixpoint: $v \preceq u$ iff v is added to $\mathrm{attr}_i(G, A)$ no later than u. (It is a preorder, because breadth-first computation of the attraction may add multiple vertices simultaneously.) A transition (u, v) between two vertices of $\mathrm{attr}_i(G, A)$ is in T_1 iff $v \prec u$.

Game solving algorithms based on attraction computation—not just Zielonkas's—therefore only compute a subset of all memoryless strategies. Even in algorithms not explicitly based on attraction computation like the one of [8], strategy computation relies on the order in which vertices are added to the set of winning positions to select transitions. Even though we have used Zielonka's algorithm for the purpose of illustration, this observation applies more generally.

III. Solving Games Incrementally

A player wins a game with a conjunctive generalized parity winning condition iff it has a strategy that simultaneously works for each parity condition. If one could compute all winning strategies for one parity condition, one would then be able to play the conjunctive game incrementally by initializing the set of candidate strategies to all possible strategies and then successively removing all strategies that are not winning for each parity condition. The surviving strategies would be winning for the conjunctive game.

An incremental approach is especially attractive when symbolic algorithms are used to solve the games, because it allows the game graph to be restructured between steps. However, the simple scheme outlined above does not work because algorithms for parity games only compute a subset of the winning strategies. For instance, algorithms for parity games only compute a subset of the memoryless winning strategies, as shown in the previous section. In this section, we show that the subset of strategies computed is not sufficient to solve conjunctive generalized parity games in an iterative fashion. We prove, however, that it is possible to decompose the solution in stages when some of the parity conditions correspond to safety or persistence properties.

Consider the game graph shown in Fig. 3 with the conjunctive winning condition $\{\pi_1, \pi_2\}$, where π_1 assigns color 1 to b and color 0 to the other vertices, while π_2 assigns color 1 to c and color 0 to the other vertices. Player 1 moves from vertex a. (That is, $Q_0 = \{b, c\}$ and $Q_1 = \{a\}$.) Note that the parity conditions are effectively Büchi conditions corresponding to the LTL formula $(\mathsf{G}\,\mathsf{F}\,b) \wedge (\mathsf{G}\,\mathsf{F}\,c)$.

Suppose the algorithm of Fig. 1 is used to compute the winning positions according to π_1. The attractor of $\{b\}$ for Player 1 is computed as follows at Line 5. Initially b is in the attractor and the set of transitions is empty; a is then added to $\mathrm{attr}_1(\{b\})$ and the transitions from a to b is added to $\Delta_1(\{b\})$. Finally c is added to the attractor and the transitions from c to a is added to the set of transitions. Since $G \backslash \mathrm{attr}_i(A)$ is empty, the recursive call returns immediately with $U_j = \emptyset$. Therefore all positions are winning for Player 1. The transition from b to a is added to $\Delta_1(\{b\})$ to produce T_1. It is clear that there is no winning strategy for π_2 when only transitions from T_1 are allowed from vertices in Q_1, though there exist strategies to win both π_1 and π_2 on G. One such strategy uses one bit of memory to alternate between the two transitions out of a.

Though in general one cannot solve generalized parity games by incrementally applying Zielonka's algorithm—one must resort instead to algorithms that can deal with multiple winning conditions like the ones of [9]—there are cases of practical relevance when it is possible to decompose the computation, namely when some of the winning conditions correspond to *safety* or *persistence* properties [23].

Definition 1. A *safety (or Π_1^0) winning condition* for a game graph $G = (Q, E)$ is a function $\pi : Q \to \{0, 1\}$ such that there is no transition $(u, v) \in E$ such that $\pi(u) = 0$ and $\pi(v) = 1$.

The winning plays according to such a condition form a closed set of the product topology of Σ^ω [24] recognized by a finite automaton. Hence, they form an ω-regular safety property. Closed sets form the Π_1^0 class in the Borel hierarchy. Conversely, an ω-regular safety property is accepted by a DPW with a safety winning condition [25].

While most translators guarantee that a syntactically safe property [17] expressed in LTL is translated into a DPW with a safety acceptance condition (a safety DPW), pathological safety properties [26] may not be recognized as such. This, however, is not a significant drawback, because such properties are quite rare in practice.[1]

One may extend Definition 1 to parity conditions with more than two colors by requesting that no transition exists connecting a state with an even color to a state with an odd color. However, when the parity conditions are of minimal index, the extended definition coincides with the one we have given.

Definition 2. A *persistence (or Σ_2^0) winning condition* for a game graph $G = (Q, E)$ is a function $\pi : Q \to \{1, 2\}$.

The winning plays according to a persistence winning condition form a regular Σ_2^0 set in the Borel hierarchy. Many commonly-used properties are recognized by weak automata. A weak automaton is a Büchi automaton such that every strongly connected component of the state graph is either included in the accepting states F or has null intersection with it [27]. It is known that nondeterministic weak word automata (NWW) and deterministic co-Büchi word automata (DCW) are equally expressive [28]. Therefore, an NWW translates into a DPW with a persistence acceptance condition. Since $\Pi_1^0 \subset \Sigma_2^0$, we could limit our discussion to persistence winning conditions; however, practical reasons that will become apparent suggest that we distinguish the two cases.

[1] Even mildly pathological cases like $(p\,\mathsf{U}\,q) \vee \mathsf{G}\,p$ are usually recognized as safety properties by translators.

A conjunctive winning condition that contains safety or persistence winning conditions can be simplified thanks to the following observation.

Theorem 1. *Let $\Pi = \{\pi_1, \ldots, \pi_k\}$ ($k > 1$) be a conjunctive winning condition for game graph $G(Q, E)$. Suppose that π_k is either a safety winning condition or a persistence winning condition and that the largest odd color in the co-domain of π_{k-1} is m. Then Π is equivalent to $\Pi' = \{\pi_1, \ldots, \pi'_{k-1}\}$, where, for each $q \in Q$,*

$$\pi'_{k-1}(q) = \begin{cases} m+1 & \text{if } \pi_k(q) \text{ is even} \\ \pi_{k-1}(q) & \text{otherwise} \ . \end{cases}$$

Proof: We reduce the case of a safety condition to that of a persistence condition by observing that the lack of paths from states of priority 0 to states of priority 1 implies that a safety winning condition $\pi_k : Q \to \{0, 1\}$ is equivalent to $\pi'_k : Q \to \{1, 2\}$ such that $\pi'_k(q) = 2$ iff $\pi_k(q) = 0$. Consider now the case of a persistence condition π_k. A play is winning according to π_k iff it visits states of priority 2 finitely many times. Therefore, a play is winning according to π'_{k-1} iff it is winning according to both π_k and π_{k-1}. ■

Repeated application of Theorem 1 eliminates all safety and persistence winning conditions (except one if there are no other winning conditions) with a maximum increase of one color in one of the surviving winning conditions (if all remaining winning conditions have an odd color as maximum priority). The "classical" algorithm of [9] runs in

$$O(m \cdot n^{2d}) \cdot \binom{d}{d_1, d_2, \ldots, d_k}, \tag{1}$$

where $n = |Q|$, $m = |E|$, $\pi_i : Q \to [d_i + 1]$ is the i-th component of the winning condition, k is the number of components, and $d = \sum_{1 \leq i \leq k} d_i$. The simplification of the conjunctive condition afforded by Theorem 1 improves the bound by reducing k and, in most cases, also d. In addition, Theorem 1 suggests a way to play conjunctive generalized parity games incrementally whenever some components of the winning condition are in Σ^0_2.

The key observation is that the winning positions according to a persistence winning condition can be divided into *persistent* and *transient*. In forming a winning strategy, transitions into persistent states are subject to no restrictions, while transitions into transient states may only be used finitely many times. The computation of persistent and transient states is described by the following theorem.

Theorem 2. *Zielonka's algorithm applied to a parity game with a persistence winning condition returns the winning positions of Player 1 as*

$$U_1 = \mathrm{attr}_1(G \setminus \mathrm{attr}_0(G, A_2)) \ . \tag{2}$$

Moreover, persistent and transient states are given by:

$$U_1^p = Q \setminus \mathrm{attr}_0(G, A_2) \tag{3}$$

$$U_1^t = U_1 \setminus U_1^p \ . \tag{4}$$

Proof: From the pseudocode of Fig. 1, it is clear that when $d = 2$ and no state has priority 0, the winning positions for Player 1 are given by (2). Moreover, all the states in U_1^p have priority 1 and from none of them can Player 0 force the visit of a state of priority 2. Therefore, a play that moves to a persistent state can always be extended to a winning play. By contrast, a winning play can only visit a state $q \in U_1^t$ finitely many times because either q has priority 2, or from it Player 0 can force the visit of another state in U_1^t with priority 2. ■

Note that the computation of the strategies in Fig. 1 is not necessary when we are only interested in determining persistent and transient states. If there is at least another component of the winning condition, it suffices to compute U_1 and U_1^t. The states not in U_1 are removed from the game graph. Those in U_1^t are given priority $m+1$ as in Theorem 1. The resulting game is equivalent to the given one.

For safety properties, which are special cases of persistence properties, the set of transient states is empty. All that is needed is U_1, which Zielonka's algorithm computes as $Q \setminus \mathrm{attr}_0(G, A_0)$. No modification of other components of the winning condition is necessary. This special case was already discussed in [29].

Since safety properties often form the bulk of a specification, and they are usually translated into safety DPWs for synthesis, the ability to treat them incrementally is significant.

Several persistence winning conditions are easily composed into one thanks to Theorem 1. Therefore, the persistence conditions can be played one at the time, all at once, or in any way in between. By playing the persistence conditions one by one, we can optimize the game graph after each game. This is our current approach. Effective heuristics for clustering the persistence conditions, on the other hand, may result in good state encodings and further speed up processing.

If there are both persistence and non-persistence winning conditions, the game is solved in two stages. First, one game is solved for each persistence condition; at each step, the game graph is pruned by removing all vertices that are losing for Player 1 and all transitions into and out of them; the winning conditions are updated accordingly. Then the game on the pruned graph defined by the remaining winning conditions is solved with an algorithm for conjunctive generalized parity games. This approach is sound because any strategy consistent with T_1 is a winning strategy for Player 1, and is complete thanks to Theorem 2. If there are no persistence winning conditions, or all conditions are persistence, one of the two phases is skipped.

The complexity of "classical" algorithm of [9] is given in (1). If π_k is a safety condition, solving the game in two stages leads to a better bound for the second stage, $O(m \cdot n^{2d-2}) \cdot \binom{d-1}{d_1, \ldots, d_{k-1}}$, while the first stage runs in $O(m \cdot n^2)$. In practice, in the second stage, the number of transitions may decrease, and the removal of losing positions for π_1 may reduce the number of colors in the remaining conditions. This may further speed up execution. Similar considerations apply when π_k is a persistence condition.

Returning to the example of Fig. 3, if π_2 is changed so as to assign color 1 to a and color 0 to the other vertices, then the strategy computed for π_1 works for π_2 as well. However, whenever a transition connecting two positions that are winning according to a given condition is removed, the survival

of at least one strategy that is winning for all conditions of a conjunctive generalized parity game depends in general on both the graph and all the parity conditions. In contrast, for safety and persistence properties, such transitions are never removed and a simple check suffices.

IV. Implementation

Our implementation takes as input a set of LTL properties and NBWs. In addition, it takes as input a partial model of the environment in Verilog. Each LTL property is translated into an NBW. All NBWs are converted into DPWs of minimum index. We use an extension of Wring [30] for translating an LTL formula into an NBW and then into a DPW. Safety properties are identified at this point, by checking Definition 1 on the DPWs. (Support for the efficient treatment of persistence properties that are not safety is being implemented.) The composition of the DPWs and the Verilog model define a game graph and a conjunctive generalized parity winning condition. If there are safety winning conditions, they are played one by one. The corresponding DPWs are composed one at the time. Each non-safety winning condition contributes one parity condition to the conjunctive game. Some of the properties in the specification may represent environmental assumptions. Together with the Verilog model, they specify the environment. The safety assumptions are treated like the safety properties that the system must guarantee, except that the game graph is pruned with the winning strategies for the opponent. The remaining assumptions for the environment are treated as antecedent for the non-safety part of the specification. A winning strategy for Player 1 in this game—if it exists—is a realization of the specifications. The game graph has an initial vertex, corresponding to the initial states of the various DPWs.

If the winning condition includes safety conditions, the first stage of game solving is then performed by running the "classical" algorithm of [9]. (This algorithm performs the same computation as Zielonka's when given one safety winning condition.) The winning positions of the safety game and the transitions among them are then the new game graph for the remaining conditions. If there is no safety condition, the second stage is performed directly on the original game graph. Once again, the "classical" algorithm of [9] is used to compute winning positions and a set of finite-memory strategies for Player 1. (In the common case in which there is one non-safety parity condition, the strategies are, obviously, memoryless.)

After the strategies have been computed for a game, constrained reachability analysis is performed on the game graph. The winning positions reachable from the initial vertex going only through other winning positions are computed. The remaining vertices are used as "don't care" conditions for the simplification of the game graph. In our current implementation, we eliminate state variables that have constant values over the "care states." We also eliminate state variables that are equivalent to other variables or are their complements over the care states. Finally, we eliminate whole automata whose state can be inferred from the state of the rest of the model. This elimination is carried out by checking whether a variable is dependent on others over the care states. Given the characteristic function of a set of states R, variable x is dependent iff $\forall x \,.\, R = \bot$. Elimination is restricted to whole DPWs to prevent adding dependencies among variables that may slow down the computations of Zielonka's algorithm. More powerful transformations are under study, but even this simple approach has significant positive effects.

After all games have been played, one strategy is extracted from the computed set. The algorithm outlined in [31] is used to heuristically attempt to extract one strategy with low implementation cost, which is written to the output in the form of a Verilog module.

V. Related Work

In this section we discuss approaches to game solving that promote some form of decomposition or incrementality and compare these ideas to our work. For a broader discussion the reader is referred to [7].

The separation of safety and liveness specifications is present in the approach of [3], which synthesizes from reactive(1) specifications. The safety component of the specification is assumed to be in the form

$$\bigwedge_{1 \le i \le p} \mathsf{G}\, B_i \ , \tag{5}$$

where the only temporal operator allowed in each B_i is the next-time operator X. Such component is effectively a pre-synthesized transition relation. We do not need the safety component of the specification to be already transformed into a transition relation in order to keep it separate. We also deal with arbitrary ω-regular specifications.

In [3], properties that translate into DBWs can be added to the specification by adding the acceptance conditions of the DBWs to the liveness specification that has the following shape:

$$\bigwedge_{1 \le i \le m} (\mathsf{G}\,\mathsf{F}\, J_i^1) \rightarrow \bigwedge_{1 \le i \le n} (\mathsf{G}\,\mathsf{F}\, J_i^2) \ . \tag{6}$$

In this case, however, separation of safety and liveness is not achieved.

The "Safraless" approach of [4], [5] is another example of multi-stage synthesis process. When the specification is a safety property, the universal co-Büchi tree automaton produced as intermediate stage can be made weak; this speeds up synthesis. However, the "Safraless" approach does not separate the safety component of the specification. Rather, in [5] it is extended so that synthesizing the conjunction of specifications can take advantage of the computations performed for the individual specifications. Our approach benefits from the weakness of the specification even when it applied to only some of the properties, thanks to Theorems 1 and 2.

The authors of [29] study a partial *permissivity* ordering on nondeterministic strategies. They show that safety games have maximal strategies that are memoryless, and that every game that has a maximal memoryless strategy is equivalent to a safety game. For other types of games, they define a *permissive* strategy as one that allows the behaviors of all memoryless strategies. They show how to build such permissive strategies with memory bounded by the size of the game graph. By

classifying winning states into transient and persistent, we can represent all strategies for persistent winning conditions (not just the memoryless ones) without introducing memory.

The authors of [6] propose an incremental approach in which a sequence of *good-for-games* automata is built until a winning strategy is found or the specification is proved unrealizable. The algorithm does not lend itself to efficient symbolic implementation and no experimental results were published. It is therefore difficult to assess the effectiveness of that approach.

VI. Experiments

The approach described in Sect. III–IV has been implemented in Vis [32] as an extension of the technique described in [7]. In this section we report on some preliminary experiments conducted on examples coming from [7] (Rrobin) and [33] (Amba Bus). Crane is the controller of a traveling crane.

Table I illustrates the benefits of the two-stage approach to playing the generalized parity games described in Sect. III and compares it to the approach of [33]. For each model, the number of persistence and non-persistence properties is given for both system (S) and environment (E) in Columns 2–5. All the persistent properties in the experiments were of the safety type. The next group of three columns reports the number of memory bits in the solution, the number of bits reported by Anzu, and the number of constant and equivalent bits removed by the optimizations applied in between games. The next three columns report the time taken by our algorithm with and without optimizations and the time taken by Anzu [33]. The last column reports the number of properties required by Anzu. Times were measured on a 2.4 GHz Quad Core Pentium Duo with 4 GB of RAM. The table shows that the advantage of incremental game playing interleaved with optimizations are quite significant for the larger examples. A monolithic specification of an 8-client Amba Arbiter, where all the properties were combined together, was also considered, but the game play did not complete in three hours; playing the game incrementally, however, finished in under 900 seconds.

The comparison with Anzu reveals several interesting differences. The number of properties required to specify the same system is much smaller in our approach. This reflects the fact that our two-stage approach accepts a more general form of specification and does not require one to pre-synthesize the safety part of the specification. The input properties to our tool and Anzu are identical except for the fact that if a safety property requires memory to be realizable then Anzu requires that the property be broken down into sub-properties and partially synthesized by hand. Table II compares two properties from Amba Arbiter specification; one that is identical for both tools; the other that needs decomposition and partial pre-synthesis for Anzu. Thanks to incremental synthesis and the optimizations that it allows, our synthesis times are quite competitive. Our approach does, however, pay a price for the more abstract specification in terms of the number of state variables in the solution. More optimizations, and in particular, a careful re-encoding of the states are required to produce more efficient implementations. The number of latches represents a crude measure of the magnitude of the synthesized model.

VII. Conclusions

We have shown that in the game-theoretic approach to the synthesis of reactive systems it is possible to separate the safety and persistence component of the specification. This allows one to solve the generalized parity game into which the specification is translated into two stages. In the presence of persistence and safety properties—a common occurrence—one therefore achieves an improved bound on the runtime of the game solving procedure. The separation in two stages also opens the door to optimizations of the game graph in between the two stages only the most straightforward of which are currently exploited by our implementation. Though these optimizations may not affect the worst-case complexity, they are practically significant and allow our algorithm to outperform in speed the best known synthesis algorithms in spite of accepting a more general and far less detailed specification. Analysis of our initial results has suggested numerous optimizations that should further increase the speed of synthesis and the quality of the resulting reactive system.

Acknowledgment

We thank Barbara Jobstmann for her help with Anzu and the Amba bus models.

References

[1] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. Symposium on Principles of Programming Languages (POPL '89)*, 1989, pp. 179–190.

[2] N. Piterman, "From nondeterministic Büchi and Streett automata to deterministic parity automata," in *21st Symposium on Logic in Computer Science*, Seattle, WA, Aug. 2006, pp. 255–264.

[3] N. Piterman, A. Pnueli, and Y. Sa´ar, "Synthesis of reactive(1) designs," in *7th International Conference on Verification, Model Checking and Abstract Interpretation*. Springer, 2006, pp. 364–380, LNCS 3855.

[4] O. Kupferman and M. Y. Vardi, "Safraless decision procedures," in *Foundations of Computer Science*, Pittsburgh, PA, Oct. 2005, pp. 531–542.

[5] O. Kupferman, N. Piterman, and M. Y. Vardi, "Safraless compositional synthesis," in *Eighteenth Conference on Computer Aided Verification*, 2006, pp. 31–44, LNCS 4144.

[6] T. A. Henzinger and N. Piterman, "Solving games without determinization," in *15th Conference on Computer Science Logic*, Szeged, Hungary, Sep. 2006, pp. 394–409, LNCS 4207.

[7] S. Sohail, F. Somenzi, and K. Ravi, "A hybrid algorithm for LTL games," in *Verification, Model Checking and Abstract Interpretation*, San Francisco, CA, Jan. 2008, pp. 309–323, LNCS 4905.

[8] M. Jurdziński, "Small progress measures for solving parity games," in *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science*. Lille, France: Springer, Feb. 2000, pp. 290–301, LNCS 1770.

[9] K. Chatterjee, T. A. Henzinger, and N. Piterman, "Generalized parity games," in *10th International Conference on Foundations of Software Science and Computation Structures*. Springer, 2007, pp. 153–167, LNCS 4423.

[10] J. R. Büchi, "On a decision method in restricted second order arithmetic," in *Proceedings of the 1960 International Congress on Logic, Methodology, and Philosophy of Science*. Stanford University Press, 1962, pp. 1–11.

[11] A. W. Mostowski, "Regular expressions for infinite trees and a standard form of automata," in *Computation Theory*, A. Skowron, Ed. Springer-Verlag, 1984, pp. 157–168, LNCS 208.

[12] E. A. Emerson and C. S. Jutla, "Tree automata, mu-calculus and determinacy," in *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, Oct. 1991, pp. 368–377.

[13] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, Oct. 1985.

TABLE I
EXPERIMENTAL RESULTS

Model	Safety		Parity		latches			Time (s)		Time (s)	Properties
	E	S	E	S	SF	Anzu	reduced	non-optimized	optimized	Anzu	Anzu
RRobin	0	10	0	0	7	6	8	0.19	0.16	0.2	20
Crane	2	12	2	2	11	n/a	3	6.7	6.3	n/a	n/a
GenBuf2	13	19	2	3	40	18	26	7.64	4.03	0.38	51
GenBuf3	15	25	2	4	49	21	31	14.09	6.49	0.6	61
GenBuf4	17	32	2	5	55	24	36	24.49	11.04	0.99	78
GenBuf5	18	43	2	6	61	27	45	49.20	15.48	2.11	79
GenBuf6	20	52	2	7	71	29	47	215.79	28.28	4.74	96
Amba2	3	17	2	3	37	24	6	4.83	6.87	2.39	56
Amba3	4	22	2	4	42	30	9	31.80	14.2	44.67	68
Amba4	5	26	2	5	48	34	6	578.8	109.9	35.30	80
Amba5	6	31	2	6	56	39	11	345.7	139.7	224.06	93
Amba6	7	34	2	7	55	43	11	789.9	301.1	1011.7	105
Amba7	8	38	2	8	61	48	10	1955.6	965.6	1758.5	117
Amba8	9	41	2	9	67	52	12	2375.6	875.3	2034.9	129
Amba9	10	44	2	10	77	57	14	8128.7	1439.6	7861.2	141
Amba10	11	48	2	11	81	61	16	11234.6	3727.6	28319.8	153
Amba11	12	51	2	12	87	65	22	TO	3154.0	8403.3	165
Amba12	13	55	2	13	92	69	21	TO	6641.2	49138.7	177
Amba13	14	60	2	14	98	73	25	TO	32562.4	13163.4	189
Amba14	15	64	2	15	105	77	19	TO	12202.2	17104.9	200

TABLE II
PARTIAL LTL SPECIFICATION OF 2-CLIENT AMBA BUS ARBITER

Property Type	SF: Safety-First	Anzu
1. In case there are no requests, the bus is granted to *Master 0*	$\mathsf{G}((DECIDE \wedge \forall i : \neg HBUSREQi)) \rightarrow \mathsf{X}(HGRANT0))$	$\mathsf{G}((DECIDE \wedge \forall i : \neg HBUSREQi) \rightarrow X(HGRANT0))$
2. No spurious grants except to *Master 0*	$\forall i \neq 0 \,.\, \mathsf{G}(\neg HGRANTi \rightarrow (HBUSREQi \,\mathsf{R}\, \neg HGRANTi)$	$\forall i \neq 0 \,.\, \mathsf{G}(((Q = init) \wedge (HGRANTi \vee HBUSREQi)) \rightarrow \mathsf{X}(Q = init))$ $\forall i \neq 0 \,.\, \mathsf{G}(((Q = init) \wedge (\neg HGRANTi \wedge \neg HBUSREQi)) \rightarrow \mathsf{X}(Q = noGrant))$ $\forall i \neq 0 \,.\, \mathsf{G}(((Q = noGrant) \wedge (\neg HGRANTi \wedge \neg HBUSREQi)) \rightarrow \mathsf{X}(Q = noGrant))$ $\forall i \neq 0 \,.\, \mathsf{G}(((Q = noGrant) \wedge HBUSREQi) \rightarrow \mathsf{X}(Q = init))$ $\forall i \neq 0 \,.\, \mathsf{G}(((Q = noGrant) \wedge HGRANTi) \rightarrow FAIL)$

[14] P. Wolper, M. Y. Vardi, and A. P. Sistla, "Reasoning about infinite computation paths," in *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, 1983, pp. 185–194.

[15] O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, New Orleans, Jan. 1985, pp. 97–107.

[16] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing, and Verification*. Chapman & Hall, 1995, pp. 3–18.

[17] A. P. Sistla, "Safety, liveness and fairness in temporal logic," *Formal Aspects in Computing*, vol. 6, pp. 495–511, 1994.

[18] W. Thomas, "On the synthesis of strategies in infinite games," in *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1995, pp. 1–13, LNCS 900.

[19] D. A. Martin, "Borel determinacy," *Annals of Mathematics*, vol. second series, 102, pp. 363–371, 1975.

[20] W. Zielonka, "Infinite games on finitely coloured graphs with applications to automata on infinite trees," *Theoretical Computer Science*, vol. 200, no. 1-2, pp. 135–183, 1998.

[21] F. Horn, "Streett games on finite graphs," in *Workshop on Games in Design and Verification*, Edimburgh, UK, Jul. 2005.

[22] M. Jurdziński, M. Paterson, and U. Zwick, "A deterministic subexponential algorithm for solving parity games," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, Miami, FL, Jan. 2006, pp. 117–123.

[23] Z. Manna and A. Pnueli, "A hierarchy of temporal properties," in *Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Quebec, Canada, Aug. 1990, pp. 377–410.

[24] L. H. Landweber, "Decision problems for ω-automata," *Mathematical Systems Theory*, vol. 3, no. 4, pp. 376–384, 1969.

[25] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, pp. 117–126, 1987.

[26] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," in *Eleventh Conference on Computer Aided Verification (CAV'99)*, N. Halbwachs and D. Peled, Eds. Berlin: Springer-Verlag, 1999, pp. 172–183, LNCS 1633.

[27] D. E. Muller, A. Saoudi, and P. Schupp, "Alternating automata, the weak monadic theory of trees and its complexity," *Theorical Computer Science*, vol. 97, pp. 233–244, 1992.

[28] C. Löding and W. Thomas, "Alternating automata and logics over infinite words," in *Theoretical Computer Science (TCS 2000)*. Berlin: Springer-Verlag, Aug. 2000, pp. 521–535, LNCS 1872.

[29] J. Bernet, D. Janin, and I. Walukiewicz, "Permissive strategies: From parity games to safety games," *RAIRO: Theoretical Informatics and Applications*, vol. 36, no. 3, pp. 261–275, 2002.

[30] F. Somenzi and R. Bloem, "Efficient Büchi automata from LTL formulae," in *Twelfth Conference on Computer Aided Verification (CAV'00)*, E. A. Emerson and A. P. Sistla, Eds. Berlin: Springer-Verlag, Jul. 2000, pp. 248–263, LNCS 1855.

[31] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Specify, compile, run: Hardware form PSL," in *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007, Electronic Notes in Theoretical Computer Science http://www.entcs.org/.

[32] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Eighth Conference on Computer Aided Verification (CAV'96)*, T. Henzinger and R. Alur, Eds. Rutgers University: Springer-Verlag, 1996, pp. 428–432, LNCS 1102.

[33] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Automatic hardware synthesis from specifications: A case study," in *In Proceedings of the Design, Automation and Test in Europe*, 2007, pp. 1188–1193.

Synthesizing Robust Systems

Roderick Bloem*, Karin Greimel*, Thomas A. Henzinger†‡, and Barbara Jobstmann†
*Graz University of Technology, †EPFL, ‡IST Austria

***Abstract*—Many specifications include assumptions on the environment. If the environment satisfies the assumptions then a correct system reacts as intended. However, when the environment deviates from its expected behavior, a correct system can behave arbitrarily. We want to synthesize *robust* systems that degrade gracefully, i.e., a small number of environment failures should induce a small number of system failures. We define *ratio games* and show that an optimal robust system corresponds to the winning strategy of a ratio game, where the system minimizes the ratio of system errors to environment errors. We show that ratio games can be solved in pseudopolynomial time.**

I. Introduction

Suppose that a system is required to accept up to 1000 requests per second and to respond to each request within 0.1 seconds. What should the system do when request number 1001 arrives? There are several options, including terminating the system, dropping the extra request, or delaying a response. Clearly, all of these approaches satisfy the specification, but some are better than others. (Cf. [1].)

The formal functional specifications used in Design Automation typically only describe the behavior of a system in absence of environment failures. That is, (parts of) the specification are of the form $A \rightarrow G$, where A is an environment assumption and G is a guarantee. This approach leaves the behavior of the system unspecified when A is not fulfilled and neither verification tools nor synthesis tools take such behavior into account. In practice, however, the environment may fail, due to incomplete specifications, operator errors, faulty implementations, transmission errors, and the like. Thus, a system should not only be correct, it should also be *robust*, meaning that it "behaves 'reasonably,' even in circumstances that were not anticipated in the requirements specification [...]" [2].

For instance, consider the following informal specification of an arbiter. Initially, both input r (for request) and output g (for grant) are low. If the environment raises r, the system will eventually raise g. The environment is not allowed to lower r before g is raised. After g is raised, r must be lowered eventually, after which g is lowered. The obvious formalization of this specification does not have any requirements on the behavior of the system if the environment lowers a request too early. In fact, if this ever occurs, the system can act arbitrarily from that time on. This is clearly unreasonable, because the system can fulfill all its requirements even in this case. In general, of course, the system may have to fail in some way if the environment does. However, we prefer graceful degradation: the system error should increase slowly with the environment error.

This paper proposes a formal notion of robustness through graceful degradation for discrete functional safety properties: A small error by the environment should induce only a small error by the system, where the error is defined quantitatively as part of the specification, for instance, as the number of failures. Given such a specification, we define a system to be robust if a finite environment error induces only a finite system error. As a more fine-grained measure of robustness, we define the notion of k-robustness, meaning that on average, the number of system failures is at most k times larger than the number of environment failures. We show that the synthesis question for robust systems can be solved in polynomial time as a one-pair Streett game and that the synthesis question for k-robust systems can be solved using *ratio games*. Ratio games are a novel type of graph games in which edges are labeled with a cost for each player, and the aim is to minimize the ratio of the sum of these costs. We show that ratio games are positional, that the associated decision problem is in NP $\cap$ co-NP, and that they can be solved in pseudopolynomial time. They can be solved in polynomial time if the cost of a failure is assumed to be constant.

Section II fixes the notation used in the paper. In Section III, we present our framework based on error functions, and define robustness and k-robustness. In Section IV, we introduce ratio games and show how to solve them. Section V shows how to use ratio games to construct correct and robust systems. We present related work in Section VI and conclude in Section VII.

II. Preliminaries

For a word $w = w_1 \ldots$, let $|w| \in \mathbb{N} \cup \{\infty\}$ be the length of the word and let $w[..i] = w_1 \ldots w_i$ be the prefix of length i. We denote the set of all finite (infinite) words over the alphabet A by A^* (A^ω).

We consider systems with a set of input signals I and a set of output signals O. We define $AP = I \cup O$. We use the signals as atomic propositions in the specifications defined below. Our input alphabet is thus $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and we define $\Sigma = 2^{AP}$.

Moore machines: We use Moore machines to represent systems. A *Moore machine* with input alphabet Σ_I and output alphabet Σ_O is a tuple $M = (Q, q_0, \delta, \lambda)$, where Q is the set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \rightarrow Q$ is the transition function, and $\lambda : Q \rightarrow \Sigma_O$ is the output function. In each state, the Moore machine outputs a letter in

This research was supported by the Swiss National Science Foundation (Indo-Swiss Research Program and NCCR MICS) and the European Union projects ArtistDesign, COMBEST, and COCONUT.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

Σ_O, then reads a letters in Σ_I, and moves to the next state. The *run* of M on a sequence $x = x_0x_1 \ldots \in {\Sigma_I}^\omega$ is a sequence $\rho_0\rho_1 \ldots \in Q^\omega$, where $\rho_0 = q_0$ and $\rho_{i+1} = \delta(\rho_i, x_i)$. The corresponding *word* is $\lambda(\rho) = w_0w_1 \ldots \in \Sigma^\omega$, where $w_i = \lambda(\rho_i) \cup x_i$. The *language of* M, $L(M) \subseteq \Sigma^\omega$, consists of the words corresponding to the runs of M. We define $L^*(M) = L(M) \cap \Sigma^*$.

Automata: A complete deterministic *automaton* over the alphabet Σ is a tuple $A = (Q, q_0, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \to Q$ is the transition function. A *run* of an automaton A on a word $w = w_0w_1 \ldots \in \Sigma^* \cup \Sigma^\omega$ is the longest sequence $\rho(w) = \rho_0\rho_1 \ldots \in Q^* \cup Q^\omega$ such that $\rho_0 = q_0$, and $\rho_{i+1} = \delta(\rho_i, w_i)$. The *product automaton* $A = A_1 \times A_2$ of two automata is defined as usual.

A *safety automaton* $A = (Q, q_0, \delta, F)$ is a complete deterministic automaton (Q, q_0, δ) together with a set $F \subseteq Q$ of *accepting states* such that there are no edges from non-accepting to accepting states.An infinite run is *accepting* if it never leaves F. An automaton accepts a word if its run is accepting. We call the set $L(A)$ of infinite words accepted by A the *language of* A.

Specifications: We use safety automata to specify the desired behavior of a Moore machine. Given a safety automaton A, we say the Moore machine M *satisfies* A, if $L(M) \subseteq L(A)$. In our examples, we also show LTL formulas describing the discussed properties. Readers familiar with LTL [3] will find them useful, while they can be safely ignore by readers not familiar with LTL.

Single and Double Cost Automata: A *single (double) cost automaton* over the alphabet Σ is a tuple $C = (Q, q_0, \delta, c)$ consisting of a complete deterministic automaton (Q, q_0, δ) and a cost function $c : Q \times \Sigma \to \mathbb{N}$ ($c : Q \times \Sigma \to \mathbb{N} \times \mathbb{N}$, respectively) that associates to each transition a value in $\mathbb{N}$ ($\mathbb{N} \times \mathbb{N}$, resp.) called *cost*. In a double cost automaton, we use c_s and c_e to refer to the cost function of the first and the second component, respectively. The *maximal cost* is the smallest $W \in \mathbb{N}$ $\forall q \in Q, \sigma \in \Sigma : c(q, \sigma) \leq W$ ($c_e(q, \sigma), c_s(q, \sigma) \leq W$). The cost of a word $w \in \Sigma^* \cup A^\omega$, denoted by $C(w)$, is the sum $\sum_{i=0}^{|W|} c(\rho(w)_i, w_i)$, For double cost automata, we use $C_e(w)$ and $C_s(w)$ to refer to the first and second component, respectively, of the cost of the word w.

The *sum of two cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is the cost automaton $A = A_1 + A_2 = (Q, q_0, \delta, c)$, where A is the product of the automata A_1 and A_2 with costs $c = c_1 + c_2$, i.e., $c((q_1, q_2), \sigma) = c_1(q_1, \sigma) + c_2(q_2, \sigma)$. The *product of two single cost automata* $A_1 = (Q_1, q_{01}, \delta_1, c_1)$ and $A_2 = (Q_2, q_{02}, \delta_2, c_2)$ is a double cost automaton $A = A_1 \times A_2 = (Q, q_0, \delta, c)$, where A is the product of the automata A_1 and A_2 with costs $c = (c_1, c_2)$, i.e., $c((q_1, q_2), \sigma) = (c_1(q_1, \sigma), c_2(q_2, \sigma))$.

Games: A *game graph* is a finite directed graph $G = (S, s_0, E)$ consisting of a set of states S, an initial state $s_0 \in S$, and a set of edges $E \subseteq S \times S$ such that each state has at least one outgoing edge. The states are partitioned into a set S_1 of *Player-1 states* and a set S_2 of *Player-2 states*. When the initial state is not relevant, we omit it and write (S, E). A *play* $\rho = s_0s_1 \ldots \in S^\omega$ is an infinite sequence of states such that for all $i \geq 0$ we have $(s_i, s_{i+1}) \in E$. We denote the set of all plays by Ω. Given a game graph $G = (S, E)$, a *strategy* for Player 1 is a function $\pi_1 : S^*S_1 \to S$ mapping a sequence of states ending in a Player-1 state to a successor state such that for all $s_0 \ldots s_i \in S^*S_1$, we have that $(s_i, \pi_1(s_0 \ldots s_i)) \in E$. A Player-2 strategy is defined similarly. We denote by Π_1 and Π_2 the set of all possible Player-1 and Player-2 strategies, respectively. A strategy is *positional* if it depends only on the current state. We present a positional strategy π_p as a function from S_p to S. Let $\rho(\pi_1, \pi_2, s)$ denote the unique play starting at s when Player 1 plays the strategy π_1 and Player 2 plays π_2.

The value of a play is given by a *value function* $v : \Omega \to \mathbb{R} \cup \{-\infty, \infty\}$. The value of a state s under strategy π_1 and π_2, denoted by $v(\pi_1, \pi_2, s)$, is the value of the play $\rho(\pi_1, \pi_2, s)$We consider complementary objectives for the two players: Player 1 tries to minimize the value of a state and Player 2 tries to maximize it. (Note that the converse is more usual.) The Player-1 value of a state s under the strategy π_1 is $\sup_{\pi_2 \in \Pi_2}(v(\pi_1, \pi_2, s))$. A strategy π_1 is optimal for Player 1 in state s if the Player-1 value of the state s under the strategy π_1 is minimal. The Player-2 value and Player-2 optimal strategies are defined correspondingly. The value of a state s denoted by $v(s)$ is the Player-1 value of the play starting in s, in which both players play optimally.

A *game* is a game graph together with a value function. The game graph defines the possible actions of the players. The value function describes the objectives of the players. A *mean payoff game* is describe as a tuple $((S, s_0, E), w)$, where (S, s_0, E) is a game graph and $w : E \to \mathbb{N}$ is a *payoff function*. The value function for a play $\rho = s_0s_1 \ldots$ in a mean payoff game is $v(\rho) = \limsup_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n} w(e_i)$ with $e_i = (s_i, s_{i+1})$. A *one-pair Streett game* is a tuple $((S, s_0, E), F_1, F_2)$ consisting of a game graph (S, s_0, E) and two sets $F_1, F_2 \subseteq S$, where $v(\rho) = 0$ iff $(\forall i \geq 0\ \exists j \geq i : s_j \in F_1) \to (\forall i \geq 0\ \exists j \geq i : s_j \in F_2)$. We say a Streett game is *winning for Player 1* if the value of the initial state s_0 is 0.

An automaton $A = (Q, q_0, \delta)$ over the alphabet Σ can be translated into a game graph (S, s_0, E) as follows. We define the set of Player-1 states as $S_1 = \{s_{(q,\sigma_i)} \mid q \in Q \text{ and } \sigma_i \in \Sigma_I\} \cup \{s_0\}$. The Player-2 states S_2 are given by the set $S_2 = \{s_{(q,\sigma_o)} \mid q \in Q \text{ and } \sigma_o \in \Sigma_O\}$. The set of game states is the set $S = S_1 \cup S_2$. Every state of the game (except for the initial state) represents a state of the automaton and an input or output label. Note that this corresponds to moving from a transition-labeled to a state-labeled system. Every outgoing transition of a state q in A is translated into two steps of the game: first, Player 1 chooses a letter σ_o from Σ_O by moving to the states $s_{(q,\sigma_o)}$, then Player 2 chooses a letter σ_i from Σ_I and moves according to the transition relation to a new state $s_{(q',\sigma_i)}$ such that $\delta(q, \sigma_o \cup \sigma_i) = q'$. Formally, we have that $E_1 = \{(s_{(q,\sigma_i)}, s_{(q,\sigma_o)}) \mid q \in Q, \sigma_o \in \Sigma_O, \text{ and } \sigma_I \in$

$\Sigma_I\} \cup \{(s_0, s_{q_o,\sigma_o}) \mid \sigma_o \in \Sigma_O\}$, $E_2 = \{(s_{(q,\sigma_o)}, s_{(q',\sigma_I)}) \mid q, q' \in Q, \sigma_o \in \Sigma_O, \sigma_I \in \Sigma_I,$ and $\delta(q, \sigma_o \cup \sigma_i) = q'\}$, and $E = E_1 \cup E_2$.

III. Defining Robustness

In this section we introduce our notion of robustness based on error specifications. We show how error specifications relate to classical specifications and the notion of realizability. We conclude with an example.

Definition 1. *An* error function *is a function* $d : \Sigma^* \cup \Sigma^\omega \to \mathbb{N} \cup \{\infty\}$. *The function is monotonically increasing in the sense that if* w' *is a prefix of* w *then* $d(w') \leq d(w)$.

An error specification *is a pair of error functions* (d_e, d_s).

The error functions define a distance between allowed and observed behavior, for instance, by measuring the number of failures in some appropriate sense. Thus, $d(w) = 0$ indicates that w fulfills the specification, and a higher value indicates a more serious violation of the specification. Error specifications provide a measure of "badness" for both the environment behavior (using d_e) and the system behavior (using d_s) and form the specifications we use in the sequel. We assume that these specifications are provided by the user.

Definition 2. *A Moore machine* M realizes *an error specification* (d_e, d_s) *if* $\forall w \in L(M) : d_e(w) = 0$ *implies* $d_s(w) = 0$.

Thus, an error specification induces a classical specification $A \to G$, where $A = \{w \in \Sigma^\omega \mid d_e(w) = 0\}$ and $G = \{w \in \Sigma^\omega \mid d_s(w) = 0\}$ are sets of infinite words.

The following notion is an alternative to realizability, forbidding the system to make mistakes before the environment does.

Definition 3. *A Moore machine* M strictly realizes *an error specification* (d_e, d_s) *if* $\forall w \in L^*(M) : d_e(w[..|w|-1]) = 0$ *implies* $d_s(w) = 0$. *An error specification is* strictly realizable *if there exists a Moore machine that strictly realizes it.*

Example 4. *An example of a specification that is realizable but not strictly realizable is* $A_1 \wedge A_2 \to G_1 \wedge G_2$, *where* x *is an input,* y *is an output,* A_1 *requires that* x *is always true* $(\mathsf{G}\, x)$, A_2 *says that* x *is initially equal to* y $(x \leftrightarrow y)$, G_1 *states that* y *is always true* $(\mathsf{G}\, y)$, *and* G_2 *states that* x *in the first step and* y *in the second step are different* $(x \not\leftrightarrow (\mathsf{X}\, y))$. *All Moore machines that realize the specification start with setting* y *to false, which violates the guarantees but forces the environment to do the same*[1].

Definition 5. *A Moore machine* M *is* robust *with respect to an error specification* (d_e, d_s) *if* $\forall w \in L(M) : d_e(w) \neq \infty$ *implies* $d_s(w) \neq \infty$.

This means that a robust system can recover from a finite environment error. Note that a system can be robust with respect to a specification that it does not realize if it contains a word with a finite system error but no environment error. Error specifications can forbid words by assigning infinite system costs. (In particular, this is possible when such specifications are given by double cost automata, as below.)

In order to calculate the quality of a robust system we want to calculate the largest system error for every environment error.

Definition 6. *A Moore machine* M *is* k*-robust with respect to an error specification* (d_e, d_s) *if* $\exists d \in \mathbb{N} : \forall w \in L^*(M) : d_s(w) \leq k \cdot d_e(w) + d$.

Obviously, every k-robust system is robust, regardless of k. Also, every robust system is k-robust for some finite k, see Theorem 15, i.e., for every finite Moore machine, the growth of the system error is either linear with respect to the environment error or unbounded. This motivates our choice of the robustness measure as a linear function. The definition of k-robustness allows us to rank Moore machines with respect to error specifications: A smaller k is better, it means that the system error increases slowly with the environment error. The constant d allows the system finitely many system failures independent of the environment error. In this paper, we focus on the infinite behavior of a machine, and note that d can be bounded by the product of the size of the Moore machine and the maximal weight. We leave minimization of d to future work.

Definition 7. *A Moore machine* (k-)robustly (and strictly) realizes *an error specification if it (strictly) realizes the specification and it is (*k*-)robust with respect to the specification.*

In the remainder, we use double cost automata to define error specifications. The environment (system) error function associated with C maps each $w \in \Sigma^* \cup \Sigma^\omega$ to its cost $C_e(w)$ ($C_s(w)$, respectively). Note that a double cost automaton can be seen as the product of two single cost automata. We can construct an error specification from a set of cost automata C_{A_i} for the system and C_{G_i} for the environment. The error specification (a double cost automaton) is the product of the sum of all C_{A_i} and the sum of all C_{G_i}.

Example 8. *Consider a system with two request signals* r_1 *and* r_2 *as inputs and two grant signals* g_1 *and* g_2 *as outputs. We want the system to respond to each request with a grant in the next step. Formally, we require that the system satisfies* $G_i = \mathsf{G}(r_i \to \mathsf{X}\, g_i)$ *for* $i \in \{1, 2\}$. *The system should also guarantee that grants are mutually exclusive, i.e.,* $G_3 = \mathsf{G}\, \neg(g_1 \wedge g_2)$. *To avoid a contradicting specification, we assume that requests are also mutually exclusive, i.e.,* $A = \mathsf{G}\, \neg(r_1 \wedge r_2)$. *Figure 1 shows two safety automata, one for* A *and one for* G_1 *and* G_2. *Note that we summarize labels on edges with Boolean expressions over* r_i *and* g_i, *where a horizontal alignment of two variables represents a conjunction and two vertically aligned variables are disjoint. We use a bar to denote negation and* $\top$ *to denote true. States depicted with two cycles are accepting states. Note that the automaton for* G_3 *is exactly the same as for* A, *where* r_1 *and* r_2 *are renamed to* g_1 *and* g_2, *respectively.*

[1]This specification is based on an example by Marco Roveri.

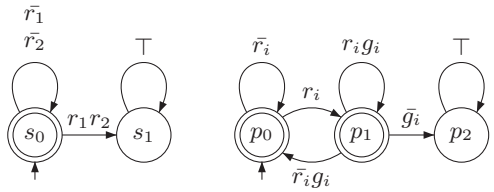
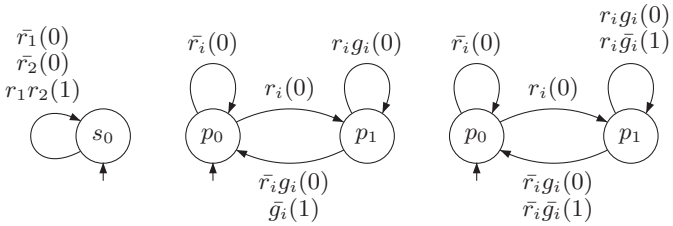


Fig. 1. Automata for $A = \mathsf{G}(\neg(r_1 \wedge r_2))$ and $G_i = \mathsf{G}(r_i \rightarrow \mathsf{X}\, g_i)$.



(a) C_A (b) C_{G_i} (c) C'_{G_i}

Fig. 2. Cost automata counting violations of A and G_i, respectively.

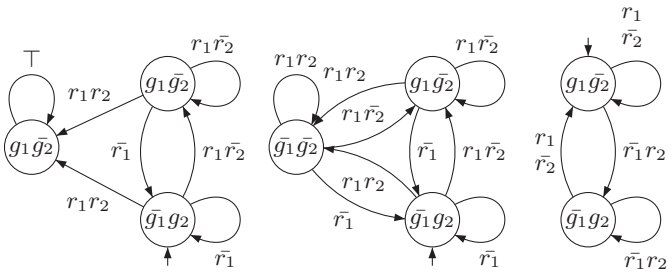
Starting from the specification $A \rightarrow (G_1 \wedge G_2 \wedge G_3)$, we can define what it means for the system and the environment to fail. In particular, the environment violates assumption A if it raises r_1 and r_2 at the same time. This corresponds to taking the edge from s_0 to s_1. In Figure 2(a), we show a cost automaton that counts every violation of the environment. Note that once the environment "pays" for taking the edge r_1r_2, we go back to the initial state, resetting the specification. Similarly, if the system violates Guarantee G_i by choosing to go from p_1 to p_2, it also incurs cost 1 as shown in Figure 2(b).

Note that it is up to the user to define the cost of a violation and the state in which to continue after the specification is violated. A reset or a skip are two natural alternatives. A reset corresponds to an edge to the initial state. For a skip, we simply add a self-loop. In Figure 2(c) we show an alternative cost automaton for G_i with $i \in \{1, 2\}$, which uses a mixture of reset and skip. For the cost automaton C_{G_i}, the word $(r_1, \bar{g_1})(r_1, \bar{g_1})(\bar{r_1}, \bar{g_1})^\omega$ has cost 1 whereas it has cost 2 for the cost automaton C'_{G_i}. For the second automaton, the cost corresponds to the number of unanswered requests.

The costs on the edges are given by the user. For instance, the user might consider a violation of the mutual-exclusion properties G_3 more severe and associate with it a higher cost than a violation of the response properties G_1 or G_2.

Given cost automata C_{G_1}, C_{G_2}, and C_{G_3} that describe the cost and the behavior associated with a violation of the corresponding property (cf. Fig. 2), we can construct a cost automaton $C_G = C_{G_1} + C_{G_1} + C_{G_1}$ for $G = G_1 \wedge G_2 \wedge G_3$. The automaton C_G defines the error function of the system. The cost automaton for the environment C_A (cf. Fig. 2(a)) specifies the error function of the environment. The product $C = C_A \times C_G$ is the error specification.

Figure 3(a) shows a system M (synthesized with Lily [4]) for the specification $A \rightarrow G$. It is easy to see that M satisfies $A \rightarrow G$. As long as the environment satisfies A, which means that it does not provide r_1 and r_2 simultaneously, the system responds to each r_i with the corresponding g_i in the next step. However, M is not robust with respect to C: The input sequence $i = (r_1r_2)(\bar{r_1}r_2)^\omega$ has cost one, but the output of the system has cost ∞.



(a) Non-robust (b) 2-robust (c) 1-robust

Fig. 3. A non-robust, 2-robust, and a 1-robust system.

Figure 3(b) and 3(c) show two systems that are robust with respect to the error specification, for any word with finitely many environment errors the systems produce finitely many system errors. The system in Figure 3(b) is 2-robust with respect to the error specification whereas the system in Figure 3(c) is 1-robust. For the input $(r_1r_2)^\omega$ the output of the first Moore machine is $(\bar{g_1}g_2)(\bar{g_1}\bar{g_2})^\omega$ and for the second it is $(g_1\bar{g_2})^\omega$.

Note that out of the three systems in Figure 3 (which all satisfy $A \rightarrow G$) the system in Figure 3(c) is the most robust one. In our opinion, it is also the one most likely to please the designer.

In Section V we show how to synthesize (strictly) realizing robust and k-robust systems from an error specification. We also show how these notions can be verified. The next section introduces Ratio games, which are crucial to our synthesis algorithms.

IV. Ratio Games

In this section we introduce ratio games, which we need to synthesize k-robust systems. Intuitively, a system is k-robust if the ratio of the system error to the environment error is smaller than or equal to k for every word of the system. An optimal strategy for Player 1 in a ratio game minimizes this ratio.

Definition 9. *A* ratio game[2] *G is a tuple $((S, s_0, E), w_1, w_2)$ consisting of a game graph (S, s_0, E) and two weight functions $w_1, w_2 : E \rightarrow \mathbb{N}$ mapping edges to non-negative integer values. The value function for a play $\rho = s_0 s_1 \ldots \in S^\omega$ is*

$$v(\rho) = \lim_{m \rightarrow \infty} \limsup_{l \rightarrow \infty} \frac{\sum_{i=m}^{l} w_1(s_i, s_{i+1})}{1 + \sum_{i=m}^{l} w_2(s_i, s_{i+1})} \qquad (1)$$

Ratio games are a generalization of mean payoff games. If $w_2(e) = 1$ for all $e \in E$, then G is a mean payoff game. Note that the sequence of quotients for $l \rightarrow \infty$ might diverge, which requires the use of $\limsup$ or $\liminf$. We follow the definition of mean payoff games and take the $\limsup$. The outer-most limit ensures that only the infinite behavior is relevant as in the definition of k-robustness, i.e., if $\sum_{i=0}^{\infty} w_1(e_i)$ is finite, then $v(\rho) = 0$. The addition of 1 in the denominator avoids

[2]Our graph-based ratio games should not be confused with those of [5], which represent games in a normal form, enumerating all strategies. We cannot use that representation to obtain a polynomial algorithm.

division by zero. It does not influence the value of $v(\rho)$ if $\sum_{i=0}^{\infty} w_2(e_i)$ is infinite.

The *maximal weight* W in a ratio game $((S, s_0, E), w_1, w_2)$ is defined by $W = \max\{w_i(e) \mid e \in E,\ i \in \{1, 2\}\}$. Note that the value $v(\rho)$ of a play ρ, where both players play positional strategies, is in the set $V = \{0, \frac{1}{|S|\cdot W}, \ldots, \frac{|S|\cdot W}{1}, \infty\}$. Lemma 10 shows that ratio games have optimal positional strategies, which implies that it suffices to consider positional strategies and that the value of every state is in V.

Lemma 10. *Ratio games have optimal positional strategies.*

Proof: It suffices to show that the two one-player games ($S_2 = \emptyset$, respectively $S_1 = \emptyset$) have optimal positional strategies [6]. Consider a game graph G with $S_2 = \emptyset$. Take in G a simple cycle with the minimum ratio r of all simple cycles.We show that the positional strategy π_1 that goes to this simple cycle and stays within it forever is optimal. Note that the value $v(\rho)$ of the play ρ induced by the strategy π_1 is r, since the outer-most limit in Eq. 1 allows us to ignore a finite prefix of ρ. If $r = 0$, the claim trivially holds. If $r = \infty$, then in any simple cycle the sum of the weights w_2 is 0 and the sum of the weights w_1 is strictly greater than 0. This implies that all edges e on cycles have weight $w_2(e) = 0$ and in every cycle there is at least one edge e with $w_1(e) > 0$, and so any infinite play has ratio ∞. For $0 < r < \infty$, let r be $\frac{a}{b}$ for some integers $a, b > 0$ and let ρ' be an arbitrary play in the single player game. We decompose ρ' into a sequence of ratios $\frac{a_1}{b_1}, \frac{a_2}{b_2} \ldots$ by the following procedure (cf. [7]): we put the states of ρ' on a stack in the order of their appearance, once we encounter a state q that is already on the stack, we remove the sequence from the first to the second appearance of q and compute its ratio $\frac{a_i}{b_i}$. Note that the sum of the weights w_1 and w_2 in this cycle can be c_i-times larger than a_i and b_i, respectively, where c_i is some integer constant between 1 and $W \cdot |S|$. Note that the height of the stack is at most $|S|$. Due to the outer-most limit, we can ignore the part of ρ' that is always left on the stack in the computation of the value $v(\rho')$. Then, $v(\rho') = \limsup_{l\to\infty} \frac{\sum_{i=1}^{l} c_i \cdot a_i}{1+\sum_{i=1}^{l} c_i \cdot b_i}$ for some constants $0 < c_i \leq W \cdot |S|$. Since the minimum simple-cycle ratio is $\frac{a}{b}$, we know that $\frac{a_i}{b_i} \geq \frac{a}{b}$ for all $i > 0$ and together with the fact that c_i's are positive integer constants, we know that $v(\rho') \geq \limsup_{l\to\infty} \frac{\sum_{i=1}^{l} a}{1+\sum_{i=1}^{l} b}$ and therefore $v(\rho') \geq \frac{a}{b}$. The proof for Player-2 games is analogous. ■

The decision problem of a ratio (mean payoff) game is, given a ratio r (mean payoff v) decide if the value of the game is at least r (v). The decision problem for mean payoff games is in NP ∩ co-NP [7]. We show how the decision problem for ratio games can be reduced to the decision problem of mean payoff games. The reduction shows that the decision problem for ratio games is in NP ∩ co-NP. We also use this reduction to calculate the values of the states in a ratio game. The reduction is similar to that used by Lawler [8] for the reduction of ratio graphs to the minimal mean cycle problem.

Lemma 11. *Let $G_R = ((S, s_0, E), w_1, w_2)$ be a ratio game with maximal weight W. Given a ratio $\frac{a}{b}$ with $0 \leq a \leq |S|\cdot W$ and $0 < b \leq |S| \cdot W$, we can decide whether a state has value $v = \frac{a}{b}$, $v < \frac{a}{b}$, or $v > \frac{a}{b}$ in $O(|S|^3 \cdot W^2 \cdot |E|)$ time.*

Proof: We reduce the decision for the ratio game to a decision for the mean payoff game $G_{\text{MP}} = ((S, s_0, E), w)$ with payoff function $w(e) = b \cdot w_1(e) - a \cdot w_2(e)$. In the following, let v_{R} ($v_{\text{R}}(\rho)$) be the value (of run ρ) in G_{R} and similarly for v_{MP}.

We show that $v_{\text{R}} \leq \frac{a}{b}$ implies $v_{\text{MP}} \leq 0$ and $v_{\text{R}} \geq \frac{a}{b}$ implies $v_{\text{MP}} \geq 0$. The decision whether $v_{\text{MP}} < 0$, $v_{\text{MP}} = 0$, or $v_{\text{MP}} > 0$ can be made in $O(|S|^2 \cdot W' \cdot |E|)$ time, where W' is the maximal weight in the mean-payoff game [7]. We have $W' \leq b \cdot W \leq |S| \cdot W^2$, thus the decision for the ratio game can be made in $O(|S|^3 \cdot W^2 \cdot |E|)$ time.

Suppose $v_{\text{R}} \leq \frac{a}{b}$. We show that Player 1 can achieve a run of value at most 0 in G_{MP} and thus $v_{\text{MP}} \leq 0$. Let π_1 be a positional optimal Player-1 strategy for G_{R} and let π_2 be a positional optimal strategy for Player 2 in G_{MP}. Because both strategies are positional, $\rho(s_0, \pi_1, \pi_2)$ consists of a stem and a simple cycle, say $\rho = (e'_1, \ldots, e'_m) \cdot (e_1, \ldots, e_n)^\omega$. Note that $v_{\text{R}}(\rho) = \frac{\sum_{i=0}^{n} w_1(e_i)}{\sum_{i=0}^{n} w_2(e_i)}$ and $v_{\text{MP}}(\rho) = \frac{b\sum_{i=0}^{n} w_1(e_i) - a\sum_{i=0}^{n} w_2(e_i)}{n}$. Suppose $\sum_{i=0}^{n} w_1(e_i) > 0$, then, since $v_{\text{R}} \leq \frac{a}{b}$ and is thus finite, we have $\sum_{i=0}^{n} w_2(e_i) > 0$. It follows that $\frac{\sum_{i=0}^{n} w_1(e_i)}{\sum_{i=0}^{n} w_2(e_i)} \leq \frac{a}{b}$ implies $\frac{b\sum_{i=0}^{n} w_1(e_i) - a\sum_{i=0}^{n} w_2(e_i)}{n} \leq 0$. If $\sum_{i=0}^{n} w_1(e_i) = 0$, then $\frac{b\sum_{i=0}^{n} w_1(e_i) - a\sum_{i=0}^{n} w_2(e_i)}{n} = \frac{-(a\sum_{i=0}^{n} w_2(e_i))}{n} \leq 0$.

The proof that $v_{\text{R}} \geq \frac{a}{b}$ implies that $v_{\text{MP}} \geq 0$ is similar, using an optimal strategy for Player 2 in G_{R}. ■

Theorem 12. *Given a ratio game $((S, E), w_1, w_2)$ with maximal weight W, the value for every $s \in S$ can be computed in $O(|S|^3 \cdot W^2 \cdot |E| \cdot \log(|S| \cdot W))$.*

Proof: We use the decision procedure from Lemma 11 to perform a binary search on the list of possible values $V \setminus \{\infty\}$. If the ratio is greater than $|S| \cdot W$, it is infinite. There are less than $(|S| \cdot W)^2$ different ratios, thus we need at most $2 \cdot \log(|S| \cdot W)$ calls to the decision procedure. ■

Given an algorithm to find the values of the game we can use the "group testing" technique from [7] to find optimal positional strategies.

Theorem 13. *Given a ratio game $((S, E), w_1, w_2)$ with maximal weight W, positional optimal strategies for both players can be found in $O(|S|^4 \cdot \log(\frac{|E|}{|S|}) \cdot |E| \cdot \log(|S| \cdot W) \cdot W^2)$.*

All our ratio game algorithms are polynomial in the size of the game graph but pseudopolynomial in the weights. They are polynomial if $W = 1$.

V. Verifying and Synthesizing Robust Systems

This section describes the verification and synthesis algorithms for robust systems. First, we establish the correlation between the ratio in Definition 9 and k-robustness.

Any error specification C with cost functions c_e and c_s can be translated into a ratio game G. The weight functions w_1 and w_2 are given by the cost functions c_s and c_e respectively. Formally, $w_{1(2)}((s_{(q,\sigma_i)}, s_{(q,\sigma_o)})) = 0$ and $w_{1(2)}((s_{(q,\sigma_o)}, s_{(q',\sigma_i)})) = c_{s(e)}(q, \sigma_o \cup \sigma_i)$, where $(s_{(q,\sigma_i)}, s_{(q,\sigma_o)}) \in E_1$ and $(s_{(q,\sigma_o)}, s_{(q',\sigma_i)}) \in E_2$ (see Section II). Every play $\rho_G = s_0, s_{(q_0,\sigma_o)}, s_{(q',\sigma_i)}, s_{(q',\sigma'_o)} \dots$ of G corresponds to a run $\rho_C = q_0 q' \dots$ of C on $w = (\sigma_o, \sigma_i)(\sigma'_o, \sigma'_i)$.

Lemma 14. *Given a Moore machine M and an error specification C with cost function c_e and c_s, M is k-robust iff for all words $w \in L(M)$, the run $\rho(w) = q_0 \dots$ of C on $w = w_0 \dots$ satisfies*

$$v(w) = \lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} c_s(q_i, w_i)}{1 + \sum_{i=m}^{l} c_e(q_i, w_i)} \leq k. \quad (2)$$

Proof: If there exists a $d \in \mathbb{N}$ such that for all finite prefixes $w' = w_0 \dots w_n$ of w we have $\sum_{i=0}^{n} c_s(q_i, w_i) \leq k \cdot \sum_{i=0}^{n} c_e(q_i, w_i) + d$, then $\frac{\sum_{i=0}^{n} c_s(q_i,w_i)}{1+\sum_{i=0}^{n} c_e(q_i,w_i)} \leq k + \frac{d-k}{1+\sum_{i=0}^{n} c_e(q_i,w_i)}$ holds as well. This implies that $\lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} c_s(q_i,w_i)}{1+\sum_{i=m}^{l} c_e(q_i,w_i)} \leq k$ because $\limsup_{l\to\infty} \sum_{i=0}^{l} c_e(q_i, w_i)$ is either some finite value d' or infinite. In the first case, $\sum_{i=0}^{n} c_s(q_i, w_i) \leq k \cdot d' + d$ for any $n \geq 0$. Therefore, $\limsup_{l\to\infty} \sum_{i=0}^{l} c_s(q_i, w_i)$ is also finite. Then, $\lim_{m\to\infty} \limsup_{l\to\infty} \frac{\sum_{i=m}^{l} c_s(q_i,w_i)}{1+\sum_{i=m}^{n} c_e(q_i,w_i)} = 0 \leq k$. In the second case, $\lim_{m\to\infty} \limsup_{l\to\infty} \frac{d-k}{1+\sum_{i=m}^{l} c_e(q_i,w_i)}$ converges to 0.

For the other direction, consider the product CM of C and M. Then, for all $w \in L^*(M)$, $C_e(w) = CM_e(w) = \sum_{i=0}^{|w|-1} c_e(q_i, w_i)$ and $C_s(w) = CM_s(w) = \sum_{i=0}^{|w|-1} c_s(q_i, w_i)$, where $\rho_{CM}(w) = q_0 \dots q_{|w|}$ is the run of CM on w. Consider an arbitrary finite word $w \in L^*(M)$, if $|w| \leq |C| \cdot |M|$, then $CM_s(w) \leq |C| \cdot |M| \cdot W$ and $C_s(w) \leq k \cdot C_e(w) + d$ holds for any $k \geq 0$ and $d = |C| \cdot |M| \cdot W$. Otherwise, if $|w| \geq |C| \cdot |M|$, we can decompose the run $\rho_{CM}(w)$ into simple cycles $c_1, \dots, c_m$ and a simple path p consisting of the remaining nodes. (See proof of Lemma 10.) Now consider the infinite words $u_1, \dots, u_m$ that correspond to the runs leading to the cycles $c_1, \dots, c_m$, respectively, and looping there forever. We know that $u_1, \dots, u_m$ are in $L(M)$ and, due to Eq. 2, that $v(u_j) \leq k$ for all $1 \leq j \leq m$. Therefore, for every cycle, the sums of the weights c_e and c_s in the cycle, are either both 0 or their ratio is smaller or equal to k. Let $k = \frac{a}{b}$ and let $\frac{a_1}{b_1}, \dots, \frac{a_m}{b_m}$ be the ratios of the cycles whose ratio is nonzero, then $\sum_{i=0}^{|w-1|} c_s(q_i, w_i) = d' + \sum_{j=1}^{m} d_j \cdot a_j$ and $\sum_{i=0}^{|w-1|} c_e(q_i, w_i) = d'' + \sum_{j=1}^{m} d_j \cdot b_j$ for some $0 \leq d', d'' \leq |C| \cdot |M| \cdot W$ and $1 \leq d_j \leq |C| \cdot |M| \cdot W$. Using the fact that, if for all $1 \leq j \leq m$, $\frac{a_j}{b_j} \leq \frac{a}{b}$ holds then $\sum_{j=1}^{m} d_j \cdot a_j \leq \frac{a}{b} \sum_{j=1}^{m} d_j \cdot b_j$ holds, we obtain $\sum_{i=0}^{|w-1|} c_s(q_i, w_i) \leq \frac{a}{b} \sum_{i=0}^{|w-1|} c_e(q_i, w_i) + d'$, which proves that M is k-robust. ■

A. Verification

We show that any robust system is k-robust.

Theorem 15. *If a Moore machine M with n_M states is robust with respect to an error specification C with n_C states and maximal system cost W, then M is $(n_C \cdot n_M \cdot W)$-robust.*

Proof: Let CM be the product of C and M. Lemma 14 shows that M is k robust if the ratio of all runs in CM is smaller or equal to k. Since one-player ratio games are positional (Lemma 10), the largest ratio corresponds to the largest ratio of a simple cycle in CM, which cannot be larger than $n_C \cdot n_M \cdot W$ because M is robust. ■

Next, we show how to verify if a given Moore machine is robust or k-robust.

Theorem 16. *Given a Moore machine M with n_M states, and an error specification C over the alphabet Σ, with n_C states and maximal cost W, we can decide if M is robust in $O(n_C \cdot n_M \cdot \Sigma)$ time. Given a k, we can check if M is k-robust in $O(n_C^3 \cdot n_M^3 \cdot \Sigma)$ time.*

Proof: Let CM be the product of C and M. M is not robust iff CM contains a cycle that contains an edge with nonzero system cost and no edge with nonzero environment cost. This can be checked in time linear in the number of edges in CM, which is $n_C \cdot n_M \cdot \Sigma$. We have that M is k-robust if the maximum simple cycle ratio in CM is smaller or equal to k. The maximum simple cycle ratio in a graph with n states and m transitions can be found in $O(n^2 \cdot m)$ time [9], thus we can find the maximum ratio in $O(n_C^3 \cdot n_M^3 \cdot \Sigma)$ time. ■

B. Synthesis

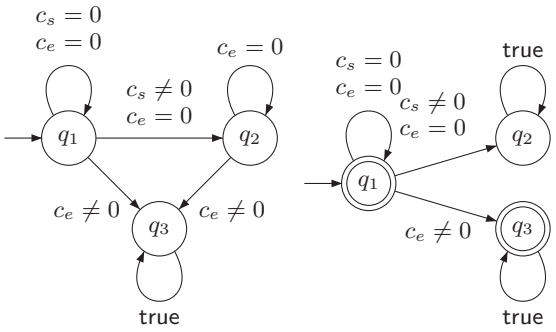
Next we show how to use Streett games to synthesize (strictly) realizing and robust systems and how to use ratio games to synthesize (strictly) realizing k-robust systems with optimal k.

Lemma 17. *Given an error specification C with n states and alphabet Σ, we can decide if a robust system exists in $O(n^2 \cdot \Sigma)$ time. If a robust system exists, it can be synthesized in $O(n^2 \cdot \Sigma)$ time.*

Proof: We translate the specification into a one-pair Streett game, F_1 is the set of states with incoming transitions with system costs and F_2 is the set of states with incoming transitions with environment costs. One-pair Streett games can be solved in $O(n \cdot m)$, where n is the number of states and m is the number of transitions [10]. ■

Theorem 18. *Given an error specification C with n states and alphabet Σ, we can decide if a robust and (strictly) realizing system exists in $O(n^2 \cdot \Sigma)$ time. The system can be synthesized in $O(n^2 \cdot \Sigma)$ time.*

Proof: In order to decide if a robust and realizing system exists, build the product automaton $CA_1 = (Q \times \{q_1, q_2, q_3\}, q_0, \delta, c)$ of the error specification C and the automaton A_1 shown in Figure 4(a). Let CA'_1 be CA_1, where the system costs of all transitions corresponding to the loop



(a) Automaton A_1. (b) Automaton A_2.

Fig. 4. Automata for calculating realizability and strict realizability

on state q_2 in Figure 4(a) are set to 1. Formally, the cost function of CA'_1 is $c'((q,x),\sigma) = (1, c_e((q,x),\sigma))$ if $x = q_2$ and $\delta((q,x),\sigma) = q_2$, and $c'((q,x),\sigma) = c((q,x),\sigma)$ in all other cases. Next, translate CA'_1 into a Streett game as above (proof of Lemma 17). A robust and realizing system exists iff the game is winning, and the winning strategy corresponds to a robust and realizing system.

First, assume there exists a winning strategy. No play in which Player 1 plays optimally visits a q_2-state infinitely often, because such a play has an infinite system error and zero environment error. Consequently, all words $w = (\sigma_o, \sigma_i)(\sigma'_o, \sigma'_i)\ldots$ associated with a play $\rho = s_0, s_{(q_0,\sigma_o)}, s_{(q',\sigma_i)}, s_{(q',\sigma'_o)}\ldots$ where Player 1 plays optimally satisfy $C_e(w) = 0$ implies $C_s(w) = 0$. Thus, the Moore machine corresponding to the winning strategy realizes the error specification. Second, assume there exists no winning strategy. A play where Player 2 plays optimally, has a finite environment cost and an infinite system cost. Either there exists no robust system or the play visits a q_2-state infinitely often. In the second case no system realizes the specification.

Similarly, to check for a robust and strictly realizing system, we build a Streett game from the product automaton CA'_2 of C and the automaton A_2 of Figure 4(b), where the system costs of all transitions corresponding to the loop on state q_2 are replaced by 1 and their environment costs are set to 0. Then, again any Player-1 optimal play avoids q_2-states. Consequently, for all words associated with a play where Player 1 plays optimally, all finite prefixes w' satisfy $C_e(w'[..|w'|-1]) = 0$ implies $C_s(w') = 0$. Thus, the Moore machine corresponding to a winning strategy strictly realizes the error specification. ■

Lemma 19. *Given an error specification C with n states, input alphabet Σ_I, output alphabet Σ_O, and maximal cost W, if a robust system exists, a k-robust system with minimal k can be synthesized in $O(n^5 \cdot (|\Sigma_I| + |\Sigma_O|)^4 \cdot \log(\frac{(|\Sigma_O|+n\cdot|\Sigma_I|)}{|\Sigma_I|+|\Sigma_O|}) \cdot (|\Sigma_O| + n\cdot|\Sigma_I|) \cdot \log(n\cdot(|\Sigma_I|+|\Sigma_O|)\cdot W)\cdot W^2)$.*

Proof: We synthesize k-robust systems with ratio games. The game graph is constructed from the double cost automaton C (see Section II). Lemma 14 shows that a positional strategy with value k corresponds to a k-robust Moore machine. An optimal positional strategy corresponds to a k-robust system with smallest possible k and $d \leq |C| \cdot W$.

The number of states in the game graph is $n\cdot|\Sigma_I| + n\cdot|\Sigma_O|$, the number of edges is $|E_1| + |E_2|$, where $|E_1| = n\cdot|\Sigma_O|$ and $|E_2| = n\cdot n\cdot|\Sigma_I|$. A winning strategy for Player 1 can be found in $O(n^4 \cdot (|\Sigma_I| + |\Sigma_O|)^4 \cdot \log(\frac{n\cdot(|\Sigma_O|+n\cdot|\Sigma_I|)}{n\cdot(|\Sigma_I|+|\Sigma_O|)}) \cdot n \cdot (|\Sigma_O| + n\cdot|\Sigma_I|) \cdot \log(n\cdot(|\Sigma_I|+|\Sigma_O|)\cdot W)\cdot W^2)$. ■

Theorem 20. *Given an error specification C with n states, input alphabet Σ_I, output alphabet Σ_O, and maximal cost W, if a robust and (strictly) realizing system exists, a k-robust system with minimal k that (strictly) realizes the specification can be synthesized in $O(n^5\cdot(|\Sigma_I|+|\Sigma_O|)^4\cdot\log(\frac{(|\Sigma_O|+n\cdot|\Sigma_I|)}{|\Sigma_I|+|\Sigma_O|})\cdot(|\Sigma_O| + n\cdot|\Sigma_I|)\cdot\log(n\cdot(|\Sigma_I|+|\Sigma_O|)\cdot W)\cdot W^2)$.*

Proof: For realizability translate CA'_1 from the proof of Theorem 18 into a ratio game. The system cost 1 for q_2-states guarantees that for any word w with $C_s(w) \neq 0$ and $C_e(w) = 0$ the ratio of the corresponding run has value ∞ in the ratio game. The ratios of other plays are not changed. If a play visits a q_2-state finitely often, the ratio is not influenced because we only look at the ratio in the limit.

For strict realizability translate CA'_2 from the proof of Theorem 18 into a ratio game. Since q_2-states have system cost 1 and environment cost 0, any run with a system failure before an environment failure has value ∞ in the ratio game.

A Moore machine corresponding to an optimal strategy of Player 1 is robust and (strictly) realizes the error specification. If k is the value of the initial state then M is k-robust. ■

C. Synthesizing from Reset Error Specifications

As shown in Example 8 *reset error specifications* are an intuitive kind of error specification. We show here that every realizable reset error specification can be realized by a 1-robust Moore machine.

Definition 21. *A* reset error specification *is a double cost automaton with maximal cost* 1*, such that for all transitions (q,σ) with $c_e(q,\sigma) = 1$ or $c_s(q,\sigma) = 1$ the next state is $\delta(q,\sigma) = q_0$.*

Theorem 22. *Given a realizable reset error specification C, a 1-robust system can be synthesized in linear time.*

Proof: Translate C into a ratio game with a linear blowup, as in Lemma 19. We show that for an optimal strategy the ratio is not greater than 1. Let π_1 be a strategy such that for all resulting plays $\rho = q_0 q_1 \ldots$, $\sum_{i=0}^{\infty} c_e(q_i, q_{i+1}) = 0$ implies $\sum_{i=0}^{\infty} c_s(q_i, q_{i+1}) = 0$. Thus, the system will not incur a cost from any state reachable using π_1 without environment cost. The only time a system cost may be incurred is when the environment incurs a cost of 1, in which case the system may also incur cost 1 and the system returns to the initial state. ■

VI. Related Work

We have defined a system to be robust if a small environment error leads to a small system error. Other approaches are possible. In the continuous domain, it is natural to require systems to be continuous, which guarantees robustness in the sense that a small output error can be guaranteed by an appropriately small input error [11]. This notion is not appropriate

in the discrete setting, as discrete functions are in general not continuous. Consider, for example, a specification that requires that the value of the output g is always true (false) if the initial input r is true (false, respectively): $(r \rightarrow \mathsf{G}\, g) \wedge (\neg r \rightarrow \mathsf{G}\, \neg g)$. Here, a minimal difference in the input, namely a change of the initial input, causes a maximal difference in the output.

The importance of robustness is widely recognized. Rinard, for instance, advocates acceptability-oriented computing, stating that "complex computer systems should have a natural resilience to errors" [12].

Attie et. al [13] argue that fault-intolerant programs are often unrealistic. They introduce a framework to specify fault-tolerant concurrent programs with CTL formulas and differnent levels of tolerance, and show how to synthesize such programs. Contrary to our work, this work considers closed systems and requires the developer to specify possible faults explicitly. Cury and Krogh consider synthesis of robust controllers for discrete event systems, where a controller is optimal if it produces the correct behavior for a maximal set of plants including the original. This approach can beneficially be combined with ours to yield systems that fulfill the guarantees in a maximal set of cases and gracefully degrade otherwise.

Faella [14] considers the question of the appropriate behavior when a game is lost. He considers two notions, one based on dominating strategies and one based on a probability distribution over the input. In the former setting, he maximizes the set of inputs for which the game is won, and in the latter setting, the probability that the game is won. A similar problem is considered in [15], where an unrealizable specification G, which corresponds to a lost synthesis game, is generalized to a specification $A \rightarrow G$ for a maximally weak environment assumption A. None of these papers considers appropriate behavior in the cases where a system failure is inevitable, which is central to our notion of graceful degradation.

D'Souza and Gopinathan [16] consider a specification that is built from a ranked set of requirements, which may be contradictory. The requirements are "conflict tolerant", i.e., when overruled, they continue giving "advice." This is achieved through means closely related to our weighted edges. D'Souza and Gopinath describe how to synthesize controllers in which advice from a requirement is alternately followed and ignored. The question they answer is how to synthesize a system that always follows the highest ranked advice. The approach differs from ours in the focus on contradictory specifications rather than environment failures, and in the fact that the proper action is chosen greedily, whereas we solve a global optimization problem to find the appropriate behavior.

Alur, Kanade, and Weiss [17], consider prioritized requirements and present an efficient way to synthesize the highest priority requirement. This is related in the sense that the ideal specification may be left unfulfilled if necessary. What is missing, from our perspective, is a way to "return" to a higher-priority requirement.

Eisner considers properties in CTL of the form $\psi = \mathsf{AG}\, \phi$ (ϕ always holds) and calls a system robust if ψ holds in all states, not just in the reachable states. This implies that the system behaves well in the presence of environment failures (assuming that any invariants used as antecedents are weak), but Eisner states that control-intensive applications are typically not robust [18].

VII. Conclusions

We have introduced a notion of robustness for functional specifications based on graceful degradation. We have shown how to solve the verification problem and the synthesis problem for robust systems. The synthesis problem is solved through a novel type of games.

We consider the worst case only: when a specification only allows for k-robust systems, we do not distinguish between systems in which every trace is strictly k-robust and those in which some traces have fewer system faults. However, Chatterjee [personal communication] has shown that admissible (undominated) strategies do not always exist for mean-payoff games, and this result easily generalizes to ratio games, foiling the hope for a fully general solution. Another venue for improvement would be to minimize the constant d in the inequality between system and environment errors. Furthermore, it is an open question how to extend our approach to liveness.

It would be interesting to evaluate to which extent our notion of robustness matches the intuitive notions designers use.

References

[1] A. M. Davis, *Software Requirements — Analysis and Specification*. Prentice Hall, 1990.

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering*. Prentice-Hall, Inc., 1991.

[3] A. Pnueli, "The temporal logic of programs," in *IEEE Symposium on Foundations of Computer Science*, Providence, RI, 1977, pp. 46–57.

[4] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, 2006, pp. 117–124.

[5] R. G. Schroeder, "Linear programming solutions to ratio games," *Operations Research*, 1970.

[6] H. Gimbert and W. Zielonka, "Games where you can play optimally without any memory," in *CONCUR*, 2005, pp. 428–442.

[7] U. Zwick and M. Paterson, "The complexity of mean payoff games on graphs," *Theoretical Computer Science*, vol. 158, pp. 343–359, 1996.

[8] E. Lawler, *Combinatorial Optimization: Networks and Matroids*. Courier Dover Publications, 1976.

[9] A. Dasdan, S. S. Irani, and R. K. Gupta, "Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems," in *DAC*, 1999, pp. 37–42.

[10] N. Piterman and A. Pnueli, "Faster solutions of Rabin and Streett games," in *LICS*, 2006, pp. 275–284.

[11] T. Henzinger, "Two challenges in embedded systems design: predictability and robustness," *Philosophical Trans. of the Royal Society*, 2008.

[12] M. C. Rinard, "Acceptability-oriented computing," in *OOPSLA Companion*, 2003, pp. 221–239.

[13] P. Attie, A. Arora, and E. A. Emerson, "Synthesis of fault-tolerant concurrent programs," *ACM Trans. Program. Lang. Syst.*

[14] M. Faella, "Games you cannot win," in *Workshop on Games and Automata for Synthesis and Validation*, 2007.

[15] K. Chatterjee, T. Henzinger, and B. Jobstmann, "Environment assumptions for synthesis," in *CONCUR*, 2008, pp. 147–161.

[16] D. D'Souza and M. Gopinathan, "Conflict-tolerant features," in *CAV*, 2008, pp. 227–239.

[17] R. Alur, A. Kanade, and G. Weiss, "Ranking automata and games for prioritized requirements," in *CAV*, 2008, pp. 240–253.

[18] C. Eisner, "Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard," in *CHARME*, 1999, pp. 97–109.

Formal Verification of Analog Designs using MetiTarski

William Denman*, Behzad Akbarpour*, Sofiène Tahar* Mohamed H. Zaki‡ and Lawrence C. Paulson§
*Dept. of Electrical & Computer Engineering, Concordia University, Montréal, Canada, {w_denm, behzad, tahar}@ece.concordia.ca
‡Dept. of Computer Science, University of British Columbia, Vancouver, Canada, mzaki@cs.ubc.ca
§Computer Laboratory, University of Cambridge, England, larry.paulson@cl.cam.ac.uk

***Abstract*—MetiTarski, an automatic theorem prover for inequalities on real-valued elementary functions, can be used to verify properties of analog circuits. First, a closed form solution to the model of the circuit is obtained. We present two techniques for obtaining the closed form solution. One is based on piecewise linear modeling and the inverse Laplace transform. The other is based on small-signal analysis and transfer function theory. Second, the properties of interest are turned into a set of inequalities involving analytic functions, which are proved automatically using MetiTarski. We verify properties concerning oscillation and the change in gain due to component tolerances.**

I. Introduction

The verification of analog integrated circuits is time consuming and requires a great deal of expertise on part of the designer. Unlike digital designs, the behaviour of analog circuits varies over continuous electrical quantities. Therefore they are highly sensitive to factors including signal noise, temperature and component variation. In addition, higher order physical effects such as parasitics and current leakage arise when designing at the submicron level. With the constant demand of shorter time-to-market, the development of computer aided and automated tools for verifying analog designs is of great importance.

Traditionally, simulation is used to verify analog designs. However, because the state space search cannot be complete, simulation methods lack the rigor to ensure the correctness of the design. By contrast, formal methods can be used to verify a model completely. Unlike in the digital domain, scalable solutions for the automated and formal verification of analog circuits remain elusive. Promising abstraction based and model checking methods have been developed where properties can be checked and counter-examples automatically generated. In particular, theorem proving can deliver the highest level of assurance for verification: an explicit formal proof.

MetiTarski [1] is an automatic theorem prover for real-valued analytical functions, including the trigonometric and exponential functions. It works by a combination of resolution inference and algebraic simplification, invoking a decision procedure (QEPCAD) [2] to prove polynomial inequalities. Its axiomatic basis consists primarily of upper and lower bounds for the special functions, obtained from their power series or continued fraction expansions. The conjecture to be proved is transformed in stages, replacing occurrences of special functions by appropriate bounds. The general resolution procedure, aided by heuristics that isolate function occurrences, accomplishes this transformation. Proofs are typically found in a few seconds [3]. MetiTarski outputs machine-readable resolution proofs, which include algebraic simplification and decision procedure calls in addition to the familiar resolution rules. These proofs, which can be checked separately, provide hard evidence for the correctness of the results.

In the last decade a new engineering field called hybrid systems has emerged. It encompasses techniques for the automatic design and analysis of systems with real-time and continuous behavior. Much work has thus been conducted on the formal verification of hybrid systems. A hybrid system can be viewed as the mathematical model of an analog circuit, which is essentially a set of differential algebraic equations. Formal methods are now a serious candidate for the verification of analog systems. In analog circuit verification, one is interested in properties connected to the dynamic behavior of the system. We are interested in properties such as: *"Will the circuit oscillate for a given set of parameters?"* and *"For all sets of constant input voltages, will switching occur in less than a specific amount of time?"*.

We demonstrate in this paper a methodology for the automatic verification of functional properties of analog designs using MetiTarski. We apply the verification methodology on two examples including a tunnel diode oscillator and an operational amplifier. The rest of the paper is organized as follows: We start with an overview of the relevant work in Sect. II. Then describe the internals of MetiTarski in Sect. III. After that we describe the verification methodology in Sect. IV. This is followed by the application examples in Sect. V. The results are shown in Sect. VI before concluding the paper with Sect. VII.

II. Related Work

The verification of analog circuits started with the work on developing finite-state discrete abstractions for computing reachability relations. Unfortunately, these methods are time bounded and computationally expensive. Greenstreet and Mitchell [4] attempted to overcome these limitations by discretizing the state space by incorporating projection techniques on the state variables. This introduces larger overapproximations but makes the verification more tractable. This allowed

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

circuits with a large state space to be verified using reachability analysis. These ideas inspired later work as in the model checking tools d/dt [5], Checkmate [6] and PHaver [7] and were respectively used in the verification of a biquad low-pass filter, a tunnel diode oscillator and a voltage controlled oscillator. Unfortunately, these three tools still rely on the use of time bounded reachability algorithms.

Another track of work has been conducted on qualitative based methods for the construction and verification of abstract models, which overcomes the time bound requirement of the reachability methods. In [8], the authors used HybridSAL [9] to generate an abstract model of several analog oscillators. Symbolic model checking was then used to prove safety properties on the generated abstract state space. The difficulty in particular with this method is that the generation of the predicates that define the abstract model is nontrivial. Human intervention is required to choose the useful and correct ones. Additionally, the abstraction can cause spurious counterexamples to be generated even if the circuit's behaviour is correct.

Our concern is the automated verification of analog circuits using deductive methods. In an early attempt at using theorem proving for the formal verification of synthesized analog circuits, Ghosh and Vermuri [10] proved the equivalence of analog designs that contain linear components and components with behaviour that can be represented by piecewise-linear (PWL) models. The PVS higher-order logic theorem prover is then used to prove the implication between implementations and behavioural specifications built in VHDL-AMS.

In similar work with theorem provers, Hanna [11] uses formal logic to define the behaviour of predicates over voltage and current waveforms. The basic behaviour of components such as resistors, power supplies and transistors are defined and then used to verify the behaviour of a NOT gate.

These early attempts are mostly based around heuristics for constructing the circuit component models and for determining the specification of the observed behaviour. Due to the underlying higher-order logic, they cannot be automated and are therefore not suited for larger applications. The methodology we present in this paper can be entirely automated and therefore could be applied to more than just basic academic problems. For information about the state of analog and mixed-signal verification, see the survey article [12].

III. MetiTarski: An Automatic Prover for Elementary Functions

MetiTarski is an automated theorem prover for real-valued special functions such as arctan, log, exp, sin, cos and sqrt. It consists of a resolution theorem prover (Metis) combined with a decision procedure (QEPCAD) for the theory of real-closed fields. Resolution works with clauses, which are typically disjunctions of inequalities, and the decision procedure assists resolution by deleting from a clause any inequalities that it finds to be inconsistent with known facts or assumptions. Deleting a literal makes progress because the aim of resolution is to generate the empty clause, which represents contradiction.

MetiTarski further depends upon being supplied with axioms approximating the functions of interest with upper or lower bounds. These approximations could be polynomials, ratios of polynomials or expressions involving other functions. For example, one axiom asserts that

$$-(x^3 + 12x^2 + 60x + 120)/(x^3 - 12x^2 + 60x - 120)$$

is an upper bound for exp(x) provided $0 \leq x \leq 4$. Each axiom will give a good approximation for some part of the real line, but typically several axioms are needed to solve a problem. Other axioms allow division (which QEPCAD does not accept) to be replaced by multiplication. The resolution proof procedure automatically tries various combinations until it is successful. A failing proof typically runs forever, though in some cases MetiTarski recognizes that no proof exists and halts with an appropriate message.

Competing methods [13], [14] typically use a combination of constraint programming and interval arithmetic. They are often powerful, but have their own limitations. They do not return proof certificates, and they require all variables to be bounded by finite intervals. They can also run forever under certain circumstances.

IV. Verification Methodology

The methodology we follow to verify analog properties is shown in Fig. 1. Starting with an analog circuit, a functional property describing some required behaviour is chosen. Using the computer algebra system Maple [15], the behavioural model of the circuit is transformed into a closed form. The property is then combined with the closed form solution and manually transformed into an inequality. The resulting expression is then processed by MetiTarski which automatically generates a proof if it can determine that the inequality holds. This resulting proof indicates that the property is true.

If MetiTarski is successful, it delivers a proof and we are done. If unsuccessful, it will run until terminated by the user. Additional axioms are then added or removed to aid MetiTarski in formulating a proof. There are certain axioms

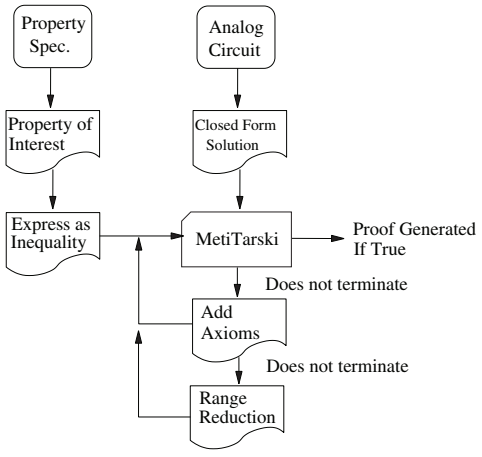


Fig. 1. MetiTarski Verification Methodology

that are available for special functions that take on extreme values. Including them unnecessarily in proofs will increase the computation time.

If still unsuccessful, an attempt at applying basic range reduction is made to the trigonometric functions to further eliminate any extreme values that can cause problems for MetiTarski's decision procedure.

There are two main difficulties: one is obtaining the closed form solution and the other is transforming the property into an inequality. The choice of property governs whether the closed form solution should be dependent on time or frequency. As well, the number and type of components in the design determine the required solution generation method.

Two closed form solution generation methods are presented and they both rely on some amount of linearization. One is based on converting the nonlinear behaviour of a component into a piecewise linear approximation. The other, similar in nature to the first, is based on linearizing the entire circuit at its DC-operating point. These methods inherently introduce some degree of error to the verification problem. In this work, we assume that the linearization is valid in the chosen neighbourhood.

A. Obtaining the Closed Form Solution: Piecewise Linearization

To obtain a closed form solution for nonlinear components, we follow the method described in Fig. 2. The idea is to separate the behaviour of the circuit in terms of its discrete "modes", such as the oscillation modes of an oscillator. Over each mode, the circuit will operate according to a different mathematical relation.

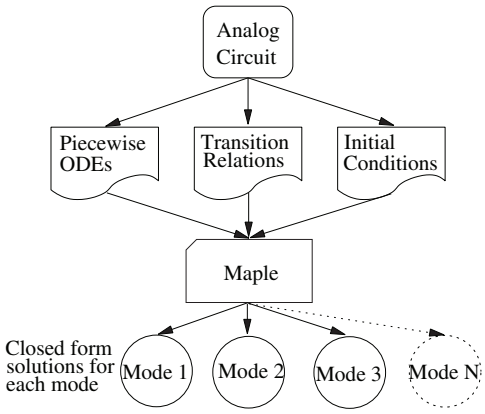


Fig. 2. Generation of the Closed Form Solution of each Mode

We first obtain the system of differential equations from the circuit of interest. Any nonlinear elements are transformed into an approximated PWL model. Due to this, there will be a certain amount of error introduced at this stage. The degree of error is set by choosing the number of segments in the PWL model. The higher number of segments, the more precise the model will be, but with an increased computing cost. Even though precision is lost with this transformation, we defend our modeling choice for the following reasons [16]:

- Piecewise-linear circuits are the simplest class of nonlinear circuits.
- The behaviour of many op-amp and diodes and switch circuits can be reasonably approximated as piecewise-linear.
- Linear methods are substantially more tractable than nonlinear ones, even when they divide the problem into multiple modes.

The transition relation between each mode of the PWL model is determined and ordinary differential equations (ODEs) are constructed over each piecewise segment. The work performed by the Maple computer algebra system is shown in Fig. 3. Starting in any mode, the ODEs and initial conditions are supplied to Maple's inverse Laplace transform routine (*invlaplace*) to calculate a closed form solution for each state variable as a function of time. Using the transition relations, the numerical solver (*fsolve*) determines the exact time where the system switches modes. At that time instance, the initial conditions for the next mode are then evaluated (*eval*) and the inverse Laplace transform is performed again to find the closed form solution. This is repeated until each mode of the model has been visited.

Note. We take the results of Maple as being correct even though no formal proof of its transformations is produced.

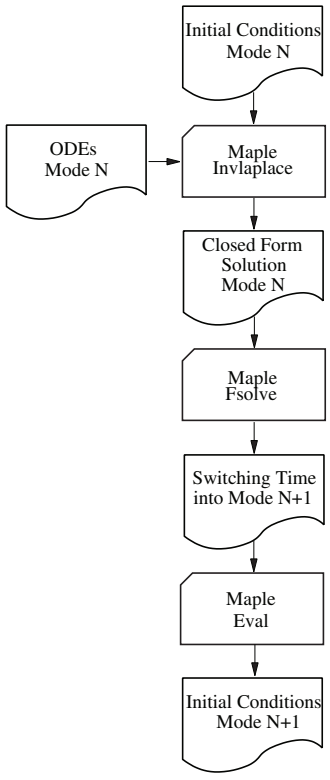


Fig. 3. Determining the Closed Form Solutions for Each Mode

B. Obtaining the Closed Form Solution : Linearization at the DC Operating Point

In the first method, we were concerned with separating the modes of operation of a single nonlinear component. This method works well when dealing with a design that contains components that operate equally over each mode of operation. When the number of nonlinear circuit elements increases, the amount of work required to keep track of the states and the transitions becomes increasingly difficult. One simplification is to assume that the components only operate over a single mode and are centered at a single voltage (DC operating point). This is the case with a transistor that operates linearly in its saturation mode of operation. By assuming that a small AC signal is superimposed on top of the DC signal, it is possible to use *small signal* analysis for calculating the closed form solution. Linearizing the entire circuit at a DC operating point greatly simplifies the generation of a single closed form solution for designs with several nonlinear components.

To obtain a closed form solution we follow the method described in Fig. 4. The circuit is linearized at its operating point. Then using circuit analysis, a transfer function that relates the input to the output is extracted from the simplified model. This method is particularly useful for generating a closed form solution that is dependent on frequency.

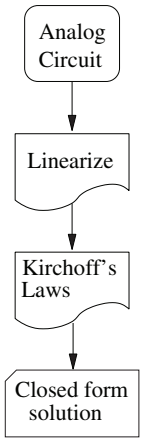


Fig. 4. Generation of the Parametric Based Closed Form Solution

C. Property Transformation

The next step is to turn the verification property into an inequality over special functions, as shown in Fig. 1. A first-order formula in the Thousands of Problems for Theorem Provers (TPTP) format, including the corresponding axioms, is then supplied to MetiTarski. MetiTarski uses an extension of the TPTP format, including infix notation for the arithmetic and relational symbols [17], [18].

There exist advanced methods to automatically extract ODEs from a circuit description. In our previous work [8], we used the Dymola modeling framework to extract simplified ODEs from a SPICE netlist. Chua and Deng [19] provide an automated method to generate the PWL model of certain op-amps, operational transconductance amplifiers and diodes. The work done with Maple is interactive and could easily be automated. The verification performed by MetiTarski is entirely automated. To show the feasibility of the proposed methodology we have applied it on several standard analog designs, two of which we present next.

V. Applications

In this section, we will describe the application of our methodology to an analog oscillator and an Operational Amplifier (Op-Amp). Oscillators play a critical role in many communication systems, in particularly for generating a periodic signal needed for the frequency translation between carriers. The tunnel diode oscillator has been previously used in [6], [20], as a benchmark for analog formal verification techniques and thus serves as an appropriate example for demonstrating our methodology. Amplifiers are the most basic component in analog circuits, which are used to control and manipulate the currents and voltages to achieve the required specifications. One of the issues with verifying such circuits is that their operation is highly dependent on process variations and therefore require many lengthy simulations.

A. Tunnel Diode Oscillator

The tunnel diode oscillator shown in Fig. 5 demonstrates the effect of resonant tunneling that causes a negative resistance to appear at small forward bias voltages as shown in Fig. 6. Essentially, for some range of voltages the current through the tunnel diode decreases with increasing voltage. This negative resistance can be used to create a reliable oscillator that functions under many different operating conditions. We intend to verify that for certain initial states and component values, the tunnel diode oscillator will not oscillate. By verifying this property, we will be able to eliminate designs that do not work.

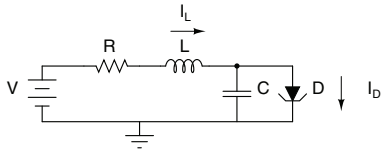


Fig. 5. Tunnel Diode Oscillator

Circuit analysis is used to determine the differential equations of the circuit. They are defined as

$$\dot{V_C} = \frac{1}{C}(-I_D(V_C) + I_L)$$
$$\dot{I_L} = \frac{1}{L}(-V_C - R \times I_L + V_{in})$$

where $I_D(V_C)$ is a PWL model that has three modes of operation. Taking E_1 and E_2 to represent the voltages where the model switches modes and G_0, G_1 and G_2 to represent the separate contributions to the slope of the best fit curve in each mode, we can define the PWL model [16] of the tunnel diode as

$$I_D(V_C) = -\frac{1}{2}(G_1E_1 + G_2E_2) + (G_0 + \frac{1}{2}G_1 + \frac{1}{2}G_2)V_D + \frac{1}{2}G_1\,|V_C - E_1| + \frac{1}{2}G_2\,|V_C - E_2|$$

Fig. 6 shows the real continuous behaviour of the tunnel diode as well as the PWL approximation. From the graph, the linearized variables are: in region 1, $g_1 = G_0$. In region 2, $g_2 = G_0+G_1$. In region 3, $g_3 = G_0+G_1+G_2$. In our example, $G_0 = 0.2616$, $G_1 = -0.3608$, $G_2 = 0.3591$, $E_1 = 0.276$ and $E_2 = 0.723$ giving

$$I_D(V_C) = \begin{cases} 0.2616V_C & \text{if } V_C < 0.276 \\ -0.0992V_C + 0.0997 & \text{if } 0.276 < V_C < 0.723 \\ 0.2599V_C - 0.1599 & \text{if } V_C > 0.723. \end{cases}$$

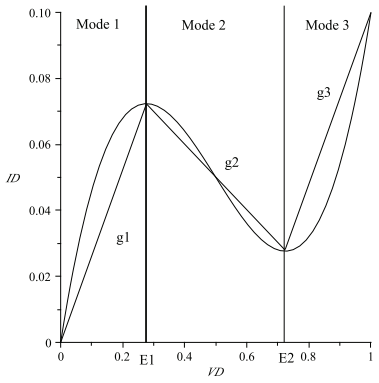


Fig. 6. Tunnel Diode Current Linearization

The system is now completely specified. Each mode is defined by a set of ODEs and switching constraints. The resulting time-deterministic hybrid model can be illustrated as an FSM as shown in Fig. 7. Each mode of operation is represented by a state circle and the switching constraints are indicated above each directional arrow.

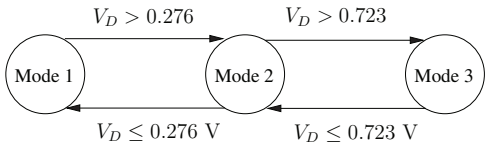


Fig. 7. The Hybrid Model of the Tunnel Diode Current

Suppose the parameter values are $R = 50\ \Omega$, $L = 10^{-6}$ H, $C = 10^{-9}$ F and $V = 0.3$ V, the dynamics of Mode 3 can be written as the first-order linear differential system $\dot{x} = Ax + B$, where the A matrix represents the coefficients of the state variables and the B matrix represents the constants.

$$x = \begin{bmatrix} I_L \\ V_C \end{bmatrix}, A = \begin{bmatrix} -3 \times 10^5 & -10^6 \\ 10^9 & -2.621 \times 10^8 \end{bmatrix}$$
$$B = \begin{bmatrix} 3 \times 10^5 \\ 0 \end{bmatrix}$$

Let X denote the Laplace transform of x (X = $\mathcal{L}x$); then $sX - x_0 = AX + \frac{B}{s}$, and solving for X we have $X = (sI - A)^{-1}(x_0) + \frac{B}{s}$. With the initial states as x_0 = (0.025,0.74)$^{\mathrm{T}}$ (a transposed matrix) and using Maple we construct the matrix as shown in Fig. 8.

The closed form solutions of the state variables are obtained by taking the inverse Laplace transform $\mathcal{L}^{-1}X$ and we obtain

$$V_C(t) = 0.116e^{-2.58\times10^8 t} + 0.278 - 0.262e^{-4.19\times10^6 t}$$
$$I_L(t) = 0.448 \times 10^{-3}e^{-2.58\times10^8 t} + 0.0727 - 0.0677e^{-4.19\times10^6 t}$$

Now we have the state space representation of the system for Mode 3 in Fig. 6. The next step is to determine the time when the tunnel diode switches from Mode 3 to Mode 2. By using Maple, we determine that the condition $V_C \leq 0.723$ is true at $t = 2.38 \times 10^{-9}$ s. The values of both V_C and I_L are evaluated at this time. We then use these values for x_0 and again repeat the process of finding the matrix X, and taking its inverse Laplace transform. This is repeated as shown in Fig. 3 until we have visited each mode and have generated the closed form solutions for the two state variables.

For Mode 2 in Fig. 6, the closed form solutions are

$$V_C(t) = 0.278 + 0.0025e^{8.79\times10^7 t} - 0.0045e^{-1.10\times10^7 t}$$
$$I_L(t) = 0.0727 + 0.00039e^{-1.10\times10^7 t} - 0.000028e^{8.79\times10^7 t}$$

For Mode 1 in Fig. 6, the closed form solutions are

$$V_C(t) = 0.323 - 01.64e^{-2.56\times10^8 t} + 0.56e^{-4.21\times10^6 t}$$
$$I_L(t) = -0.076 - 0.00064e^{-2.56\times10^8 t} + 0.144e^{-4.21\times10^6 t}$$

To demonstrate the power of MetiTarski, we seek to define an oscillation property that can be proved over all modes of operation. One such property is *"For a set of initial conditions, the circuit will not oscillate"*. In this example we focus on the current through the inductor. When the Tunnel

$$X = \begin{bmatrix} \dfrac{(s + 0.262 \times 10^9)(0.55 \times 10^{-2} + \dfrac{0.300 \times 10^6}{s})}{(s^2 + 0.262 \times 10^9 s + 0.108 \times 10^{16})} - \dfrac{0.131 \times 10^6}{(s^2 + 0.262 \times 10^9 s + 0.108 \times 10^{16})} \\ \dfrac{(0.550 \times 10^7 + \dfrac{0.300 \times 10^{15}}{s})}{(s^2 + 0.262 \times 10^9 s + 0.108 \times 10^{16})} + \dfrac{(0.131s + 0.393 \times 10^5)}{(s^2 + 0.262 \times 10^9 s + 0.108 \times 10^{16}))} \end{bmatrix}$$

Fig. 8. Tunnel Diode Matrix

Diode oscillates, the current through the inductor will range between a set of values. A necessary condition for oscillation is that the current pass some threshold. Since this requirement is not sufficient for oscillation to occur we must focus on non-oscillation. If we choose an initial point and the current does not exceed the threshold, then we can conclude that the circuit does not oscillate. This property can be more exactly defined as *"For all time and all possible paths, the current through the inductor will never pass some upper or lower bound"*. For example, when the upper bound is 0.03, the property can be expressed as:

Property 1: $[I_L \leq 0.03]$

The first order formula we supply to MetiTarski is in its TPTP-syntax. For example, to prove that in mode 1, I_L is always less than 0.03 we use the following

```
fof(
Tunnel,conjecture, ! [X] :
((0 <= X & X <= 2.39*10^(-9)) =>
 (- 0.076 - 0.00064*exp(-2.56*10^8*X)
  + 0.144*exp(-4.21*10^6*X))
  < 0.03)).
```

where *'fof'* indicates to MetiTarski that the logic language used is a first-order formula. It is then followed by a label of the proof as well as the keyword *'conjecture'* indicating that the following formula is to be proved with the included axioms. The conjecture is read as follows: For all (represented by "!") X between 0 and 2.39×10^{-9} the formula is always less than 0.03.

Now suppose we choose the component values $R = 0.3\,\Omega$, $L = 10^{-6}$ H, $C = 10^{-9}$ F, $V = 0.3$ V. Using the same inverse Laplace transform methodology, we get the closed form solutions of the state variables. The property of interest is now: *For a set of initial conditions, the trajectory of the oscillation reaches a final set and remains bounded* [21]. The variables of the circuit that oscillate are V_C and I_L. This can be described formally as:

Property 2: $[V_C > 0 \wedge V_C < 0.9 \wedge I_L > 0 \wedge I_L < 0.08]$

MetiTarski proves both properties over the three modes of operation. For property 1, it is proved that the circuit does not oscillate. For property 2, it is proved that the oscillation present in the circuit is bounded. Complete runtime results of this example can be found in Tables I and II.

B. Operational Amplifier

In this final example, a frequency domain property of a CMOS Operational Amplifier will be analyzed and verified. The Op-Amp is a popular device because of its versatility [22]. It is a fundamental building block of many designs including differential amplifiers, integrators, differentiators and digital to analog converters. One characteristic that makes verification of Op-Amps a simpler task is that its behaviour approaches the idealized model under certain operating conditions.

The analysis of the frequency domain is important since an input signal is usually not constrained to a single frequency. The performance of a device will behave differently at high frequencies. Consider the circuit in Fig. 9, as the frequency of the input signal increases, there will be a point where the gain drops below a specified level.

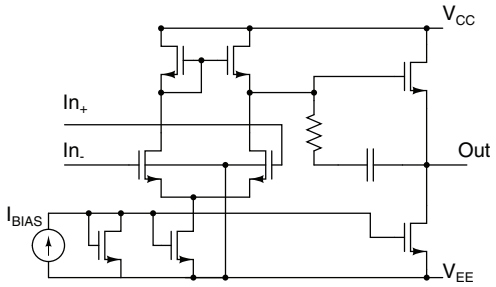


Fig. 9. Operational Amplifier [23]

To begin verification, the circuit is first linearized at its operating point and then using nodal analysis (Kirchhoff's current and voltage laws), the following transfer function is extracted

$$H(s) = \frac{A_0 e^{\Delta\Phi}}{1 + A_0 \frac{\sin(phm)}{gbw 2\pi} s + A_0 \frac{\cos(phm)-1}{gbw^2 4\pi^2} s^2} \text{[23]}$$

where phm represents the phase margin, gbw the gain bandwidth, $\Delta\Phi$ the phase tolerance and A_0 the closed loop gain. The phase margin is an indicator of amplifier stability. The phase tolerance represents the change in phase from input to output. The closed loop gain represents the gain of the Op-Amp when connected in a feedback configuration. What we would like to determine is that over a certain range of parameter values, does the gain of the circuit remain above some minimum value.

By taking the absolute value or magnitude of $H(s)$, a closed form solution for the gain of the circuit is obtained. With the closed loop gain chosen to be 93 dB and the gain bandwidth to be 5 MHz, the gain is now characterized by the equation in Fig. 10.

From the specification of the circuit [23], we are given that in the frequency range of 100 to 120 Hz the gain of the circuit should be greater than 57000 and this can be expressed as:

$$|H(jw)| = \frac{5.9 \times 10^{19}}{\sqrt{\frac{10^{29}}{0.1} - \frac{10^9 w^4 \cos(phm)}{0.28} - \frac{10^5 w^4 \cos(phm)}{0.83} - \frac{10^{20} w^2 \cos(phm)}{0.83} + w^4 + \frac{10^{24} w^2}{0.28} - \frac{10^{24} w^2 \cos(phm)^2}{0.28}}}$$

Fig. 10. Gain of the Operational Amplifier

Property 3:

$[Freqs.$100 to 120 Hz, phm 45 to 60 Deg : $|H(s)| > 57000]$

Using MetiTarski

```
fof(OPAMP,conjecture, ! [X,Y] :
((100 <= X & X <= 120 &
  PI/4 <= Y & Y <= PI/3) =>
 (5.9*10^19/
  (sqrt(9.7*10^29
   - 3.6*10^9*X^4*cos(Y)^2
   - 1.2*10^5*X^4*cos(Y)
   - 1.2*10^20*X^2*cos(Y)
   + X^4 + 3.6*10^24*X^2
   - 3.6*10^24*X^2*cos(Y)^2))
  > 5700))).
```

MetiTarski proves that the property holds over the entire frequency range. Specifically, that the gain of the circuit does not decrease below the required level. The runtime results are found in Table III.

Mode	Variable	Bound	CPU Time (sec.)
1	I_L	U	0.1
2	I_L	U	4.0
3	I_L	U	0.3

TABLE I
TUNNEL DIODE OSCILLATOR - PROPERTY 1 RESULTS

Mode	Variable	Bound	CPU Time (sec.)
1	V_C	U	0.2
1	V_C	L	0.4
2	V_C	U	2.7
2	V_C	L	0.6
3	V_C	U	0.3
3	V_C	L	0.5
1	I_L	U	0.5
1	I_L	L	0.3
2	I_L	U	0.6
2	I_L	L	3.9
3	I_L	U	0.3
3	I_L	L	0.6

TABLE II
TUNNEL DIODE OSCILLATOR - PROPERTY 2 RESULTS

Mode	Variable	Bound	CPU Time (sec.)
Saturation	$\|H(s)\|$	L	8.64

TABLE III
OPAMP - PROPERTY 3 RESULTS

VI. EXPERIMENTAL RESULTS

In the previous section we presented concrete examples of how, using the theorem prover MetiTarski, various analog circuit properties could be verified. The experimental results are presented in Tables I, II and III. In each table, the name of each experiment represents the mode (1, 2 or 3), variable under test (V_C or I_L) and either the upper (U) or lower (L) bound. For the tunnel diode oscillator we first proved in one case that oscillation is not present. The results show three experiments that indicate I_L never passes some upper bound in any mode. In the other case, it was necessary to conduct 12 experiments to prove that each of the three variables are bounded in each mode. The frequency dependent gain of an operational amplifier was also verified. The runtimes were measured on a 2.8 GHz Dual Quad-Core Mac Pro, with 4GB of RAM.

The experimental results indicate that it is possible to solve simple analog circuit verification problems using an automated theorem prover. We obtain formal proofs that can be inspected in order to increase our confidence that the design correctly matches its specification. Most of the experiments return in less than 5 seconds. For those that took longer, this is explained by the extreme values taken by the special functions of the closed form solutions. It is sometimes possible to perform range reduction to reduce the time that is necessary to complete the proof. Unfortunately, range reduction is not trivial to apply to trigonometric equations.

VII. CONCLUSION

First and foremost we have developed a viable methodology for the automated verification of analog designs. Starting with the system of equations model of the analog circuit, the closed form solutions of each mode of operation is generated using Maple. The closed form solutions are then passed to the MetiTarski theorem prover along with properties of interest defined in terms of inequalities. MetiTarski then generates a full proof of its claim of truth. Secondly, we have demonstrated that the methodology can be applied to a certain set of analog circuits. The tunnel diode oscillator analyzed in the paper has an interesting and complex behaviour that requires a high level of verification to ensure proper functionality. The results that we have obtained are promising and we are now interested in applying the methodology to different classes of circuits. The proofs we have obtained are performed quickly and this is an indication that our methodology could be scaled to more complicated problems.

To scale to larger problems, we will need to investigate efficient methods for analyzing nonlinear systems. Extensions to our work could include methods for analytically solving systems of polynomial nonlinear ordinary differential equations. One such method is the Prelle-Singer procedure [24], which is implemented in computer algebra systems such as REDUCE (the PSODE package [25]) and Maple (the PSsolver package [26]). Furthermore, the automation of the mechanical steps must be addressed. This will include an investigation on methods to automatically calculate the piecewise linear functions of nonlinear circuit elements and to automate the work performed by Maple.

We are quite motivated by the results of the work in this paper and further experimentation is ongoing. A necessary addition to the methodology would be to increase the precision of the PWL models by introducing an error bound. Indeed, we will need to apply MetiTarski to bigger and more complex examples, where a limiting factor is formulating the property of interest in terms of analytical functions.

REFERENCES

[1] B. Akbarpour and L. C. Paulson, "MetiTarski: An automatic prover for the elementary functions," in *Intelligent Computer Mathematics*, ser. LNCS, vol. 5114. Springer, 2008, pp. 217–231.

[2] C. W. Brown, "QEPCAD B: a program for computing with semi-algebraic sets using CADs," *SIGSAM Bulletin*, vol. 37, no. 4, pp. 97–108, 2003.

[3] B. Akbarpour and L. C. Paulson, "Applications of MetiTarski in the verification of control and hybrid systems," in *Hybrid Systems: Computation and Control*, ser. LNCS, vol. 5469. Springer, 2009, pp. 1–15.

[4] M. R. Greenstreet and I. Mitchell, "Reachability analysis using polygonal projections," in *Hybrid Systems: Computation and Control*, ser. LNCS, vol. 1569. Springer, 1999, pp. 103–116.

[5] T. Dang, A. Donze, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 3312. Springer, 2004, pp. 21–26.

[6] S. Gupta, B. H. Krogh, and R. A. Rutenbar, "Towards formal verification of analog designs," in *IEEE/ACM International Conference on Computer Aided Design*, 2004, pp. 210–217.

[7] G. Frehse, B. H. Krogh, and R. A. Rutenbar, "Verifying analog oscillator circuits using forward/backward abstraction refinement," in *Proc. IEEE/ACM Design, Automation and Test in Europe*, 2006, pp. 257–262.

[8] W. Denman, M. Zaki, and S. Tahar, "A bond graph approach for the constraint based verification of analog circuits," in *Workshop on the Formal Verification of Analog Circuits*, Jul. 2008.

[9] A. Tiwari, "HybridSal: A tool for abstracting hybridsal specifications to SAL specifications." `http://sal.csl.sri.com/hybridsal/`

[10] A. Ghosh, R. Vemuri, and D. R. Vemuri, "Formal verification of synthesized analog designs," in *IEEE International Conference on Computer Design*, 1999, pp. 40–45.

[11] K. Hanna, "Reasoning about analog-level implementations of digital systems," *Formal Methods in System Design*, vol. 16, no. 2, pp. 127–158, 2000.

[12] M. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs: A survey," *Microelectronics Journal*, vol. 32, no. 12, pp. 1395–1404, Dec. 2008.

[13] S. Ratschan and Z. She, "HSolver : Verification of hybrid systems based on the constraint solver RSolver." `http://hsolver.sourceforge.net/`

[14] M. Franzle and C. Herde, "HySAT : An efficient proof engine for bounded model checking of hybrid systems," *Formal Methods in System Design*, vol. 30, no. 3, pp. 179–198, Jun. 2007.

[15] "Maple 12 : The essential tool for mathematics and modelling." `http://www.maplesoft.com/`

[16] W. K. Chen, *The Circuits and Filters Handbook*. CRC Press LLC, New York, 2006.

[17] G. Sutcliffe and C. Suttner, "The TPTP Problem Library: CNF Release v1.2.1," *Journal of Automated Reasoning*, vol. 21, no. 2, pp. 177–203, 1998.

[18] ——, "The TPTP problem library for automated theorem proving," 2009. `http://www.cs.miami.edu/~tptp/`

[19] L. O. Chua and A.-C. Deng, "Canonical piecewise-linear analysis: Generalized brake point hopping algorithm," *International Journal of Circuit Theory and Applications*, vol. 14, no. 1, pp. 35–52, 1985.

[20] W. Hartong, K. Klausen, and L. Hedrich, "Formal verification for non-linear analog systems: Approaches to model and equivalence checking," in *Advanced Formal Verification*. Kluwer, 2004, pp. 205–245.

[21] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," in *Hybrid Systems: Computation and Control*, ser. LNCS, vol. 3414. Springer, 2005, pp. 263–279.

[22] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*. Oxford University Press, 2004.

[23] L. Hedrich and E. Barke, "A formal approach to verification of linear analog circuits with parameter tolerances," in *IEEE/ACM Design, Automation and Test in Europe*, 1998, pp. 649–654.

[24] M. S. M. Prelle, "Elementary first integrals of differential equations," *Transactions of the American Mathematical Society*, vol. 279, no. 1, pp. 215–229, Sep. 1983.

[25] Y. Man, "Computing closed form solutions of first order odes using the prelle-singer procedure," *Journal of Symbolic Computing*, vol. 16, no. 5, pp. 423–443, 1993.

[26] L. Duarte, S. Duarte, L. da Mota, and J. Skea., "An extension of the prelle- singer method and a maple implementation," *Computer Physics Communications*, vol. 144, no. 1, pp. 46–62, Mar. 2002.

Formal Verification of Correctness and Performance of Random Priority-based Arbiters

Krishnan Kailas
IBM T. J. Watson Research Center
Yorktown Heights, NY
kailas@us.ibm.com

Viresh Paruthi Brian Monwai *
IBM Systems & Technology Group
Austin, TX
vparuthi@us.ibm.com, bmonwai@u.washington.edu

***Abstract*—Arbiters play a critical role in the performance of electronic systems. In this paper, we describe a novel method to formally verify correctness and performance of random priority-based arbiters. We define a property of random number sequences, called Complete Random Sequence (CRS), to characterize bounded fairness properties of random number generators and random priority-based arbiters. We propose a three step verification method utilizing the notion of CRS to establish deadlock-free operation of the arbiters, and to accurately quantify the request-to-grant delays. The proposed verification method may additionally be leveraged to tune systems composed of random priority-based arbiters and pseudo-random number generators, such as linear feedback shift registers (LFSRs), for optimal performance. We have successfully applied the approach to verify a host of cache arbiters and interconnection network controllers of commercial microprocessors.**

I. Introduction

Arbiters [1] are used extensively in electronic systems such as microprocessors, interconnection networks and other peripheral chips. The primary function of an arbiter is to restrict access requests to a shared resource, such as cache directories and buses, when the number of requests exceeds the maximum number that can be satisfied concurrently. A variety of arbitration schemes are employed by arbiters to serialize the access requests based on assigning a priority to the input requests, such that the highest priority requests are granted access to the shared resources before other pending or concurrent low priority requests.

Several priority functions [1] such as fixed-priority (certain requests always have higher priority than others), round-robin priority (strict rotation of priority assignment), priority assignment based on request arrival time (first-in first-out or least recently used), and random priority (any request can have the highest priority, at random) are commonly used by arbiters. Random-priority based arbiters are often preferred in many parts of large chip designs because they provide potentially fair arbitration while using relatively less logic and power. The latter is a by-product of sharing a few on-chip pseudo-random number generators across a number of arbiters with each arbiter tapping a subset of the bits of the pseudo-random generator to assign priorities to input requests at "random".

Regardless of the arbitration scheme used, it is important to specify and verify the desired fairness properties of the arbitration scheme. Fairness ensures that all requests are granted within a finite amount of time (based on the arbiter design specifications) and no requests are forced to starve. Formal verification techniques may be used to verify that the arbitration logic is starvation-free. In addition, formal verification may be used to prove that the arbiter design adheres to the design performance requirements, such as request-to-grant delay bounds.

In this paper we focus on the formal verification of random priority-based arbiters. Our focus will be on verifying deadlock/ starvation free operation of the arbiters. Other interesting safety properties can be verified easily with suitable methods. For eg., mutual exclusion of grants can be verified by using a counter for counting the number of grants in each cycle, and using traditional bounded-model checking techniques to check for the number of grants (counter value) never exceeding the maximum number of concurrent grants allowed by the design specifications. The ideas presented in the paper may be generalized to other types of arbiters (eg. round robin, two-level random priority-based) by suitably quantifying the fairness aspects of the respective arbitration schemes.

Typical formal verification approaches used for verifying arbiters leverage temporal logics [2] to specify liveness properties [3], which are then evaluated by an underlying decision procedure to check for the presence of deadlocks/starvation. The focus of such prior techniques is to prove that deadlocks/starvation cannot occur in an infinite execution of the machine. The decision procedure analyzes the reachable states, and looks for loops in the state transition graph. However, such a proof of liveness simply guarantees that a grant will be issued eventually, and does not provide any insights into the upper bound on the number of clock cycles between a request and a grant. The grant can be delayed for an unacceptable number of cycles, making such a proof less valuable from the standpoint of the performance requirements/specification of the design. Hence, there is a need for an improved methodology to (formally) verify arbitration logics inclusive of the performance/quality-of-service aspects of the logics.

We present such a verification methodology in this paper. We articulate the absence of deadlocks as a bounded liveness check, which has several advantages. It enables precisely computing the bounds on the request-to-grant delay, giving insights into the performance of the logic, and greatly im-

* Brian Monwai is currently at the University of Washington, Seattle.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

proves computational efficiency as safety property checking is easier to evaluate algorithmically compared to liveness. It may be noted that while it is common to cast a liveness check as a bounded safety check, our proposed methodology uniquely quantifies the fairness and request-to-grant delay characteristics of the arbitration logic.

The main advantages of our verification scheme are as follows:

- The method effectively decouples the fairness logic from the actual arbitration logic, allowing checking the bounded fairness properties of each independently. Hence, the proposed scheme scales well to verification of large arbiters.
- The scheme provides a method to quantify the fairness properties of pseudo-random number generators [4] and arbiters that use random priority-based arbitration schemes.
- The approach improves upon prior art verification schemes, which guarantee deadlock-free operation of arbiters, by accurately computing the request-to-grant delay of the arbiters as well, thus enabling verification of performance aspects of the design.
- The technique can be applied to the verification of RTL directly ensuring correctness of the real logic, and does not require building any specialized models.

Organization. The rest of the paper is organized as follows. In the next section we briefly review the related work and highlight the novel aspects of our proposed solution. Section III provides a general background about random priority-based arbitration scheme. In section IV, we discuss the issues in specifying a fairness property for random number sequences, and introduce the notion of Complete Random Sequence. We describe the details of the proposed formal verification method in section V, and discuss the results in section VI, followed by some conclusions.

II. Related Work

Arbiters are routinely verified using formal verification techniques (eg. [5], [6]) such as Symbolic Model Checking [7], [8]. Typical approaches check for starvation, specified as a temporal logic liveness formula, which checks for eventual grant of resources to the requesters in an infinite execution of the machine - a property which is impossible to verify using prevalent underapproximate techniques such as simulation. Liveness properties are easy to formulate and verify against the logic, which has helped establish formal verification as the key to verifying arbiters. As stated above, such techniques do not give much insight into the performance of the arbitration logic, such as whether the requests are granted access to shared resources within a specified number of cycles.

Checking for unbounded liveness though poses a challenge due to higher computational complexity than other classes of properties. Hence, practical approaches cast the unbounded liveness check as a bounded check for liveness [9], [10] and bounded fairness [11]. This necessitates the re-tooling of the bounded liveness check to take into account fairness constraints imposed on the logic. While such an approach does allow for verification of liveness using a wider range of decision procedures (such as SAT) and gives some insights into the performance, it still is somewhat of an approximation as the fairness has to be "artificially" built into the property, and not really checked for against the actual fairness logic. Alternately, the liveness property may be converted into a safety property using transformations such as [12], [13]. This enables leveraging a rich set of verification algorithms (eg. Transformation-based [14]) to alleviate the capacity problems somewhat [15].

Our proposed ideas go beyond the above techniques by way of precisely characterizing the performance of the arbiter and the fairness logic, which can then be compared against the design specification for conformance, or used to ascertain system performance. Moreover, the insights we provide can be used to optimize the fairness logic, eg. fine tune the LFSR logic to generate all possible unique random numbers within a specified number of cycles, or select the right subset of bits to tap for optimal performance. We decompose the verification task into checks on the fairness logic and the arbitration logic separately which further addresses computational complexity of the task.

III. Random priority-based arbiters

Random priority-based arbiters are commonly used for granting a subset of several concurrent read and write requests to access a shared resource such as a cache directory or a shared bus in every cycle. In random priority-based arbitration, any request can become the highest priority request at random. For example, as shown in Figure 1, request with ID i gets its turn at time t when the value of random number $r(t) = f(i)$, where $f(i)$ is a function of i. The random numbers are usually generated using a pseudo-random number generator such as a Linear Feedback Shift Registers (LFSR) [4]. The goal of this arbitration scheme is to provide unbiased service to all requests.

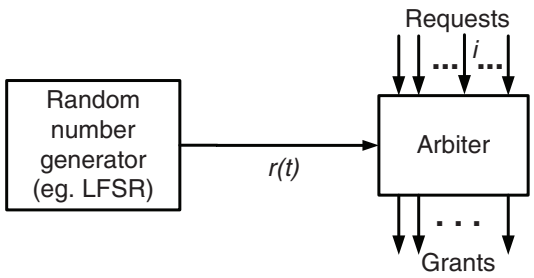


Fig. 1. Random priority-based arbiter.

In such random priority-based arbiters, one of the pending requests is granted access to the shared resource based on the random number generated by the LFSR logic in the current cycle. A request will be starved (i.e., its grant will be delayed for a long time) if the specific random number $r(t)$ that allows a request i to be granted is not generated by the LFSR logic for a long time. Clearly, delaying a request beyond a certain

number of cycles can have serious performance impact, for example, when such an arbiter is used in a cache directory access control logic. Therefore, in real applications, it is not sufficient to prove that the random number generator produces a random number sequence containing all possible values because it only proves that any given random number will be generated *eventually* (i.e., any given request will be granted *eventually*). We additionally need to define fairness properties for the random number generator in order to reason about and prove the request-to-grant delay bounds of the arbiter.

In the next section, we explain the key insights that led to our formal verification method.

IV. Specifying bounded fairness property for random number sequences

True random numbers are hard to generate. This has led to the creation of a host of pseudo-random number generators in use today with different characteristics. Random number generators, and the sequence of random numbers generated by them, are characterized mainly by a few key properties such as predictability and distribution. Predictability measures the randomness of the pseudo-random generator by characterizing occurrences (or lack thereof) of repeating sequences of random numbers generated. Distribution measures the frequency of occurrence of each number in an infinitely long sequence of random numbers generated; it is desirable to have a uniform frequency distribution, i.e., each random number must have the same frequency. However, the property of greatest importance in ensuring that a random priority-based arbiter will grant a request within an acceptable finite time interval is a different one – this fairness property must ensure that any unique random number will be generated within a fixed time interval. We use the notion of *Complete Random Sequence* to characterize the fairness property of a random number sequence, and the logic (such as LFSR) used to generate such a number sequence.

A. Complete Random Sequence

We define a Complete Random Sequence (CRS) in a random number sequence as a contiguous sequence of random numbers that has *all* the possible unique random numbers at least once. There are 2^N unique numbers in the output of a random number generator (e.g. LFSR) that can generate an N bit random number. Therefore, if the random number generator can generate one random number in every cycle, the length of the shortest CRS will be 2^N cycles. The longest CRS generated by a pseudo-random number generator can be infinitely[1] long. In other words, a random number generator may take a variable number of cycles from 2^N to infinity to generate a CRS.

V. Formal verification method

In the following we present a formal verification method that uses CRS (defined above) to specify the bounded fairness properties of random priority-based arbiters, and to accurately quantify and verify the request-to-grant delays.

[1]The CRS can be infinitely long if one (or more) of the 2^N random values is missing from the shift register sequences [16] used by the arbiter.

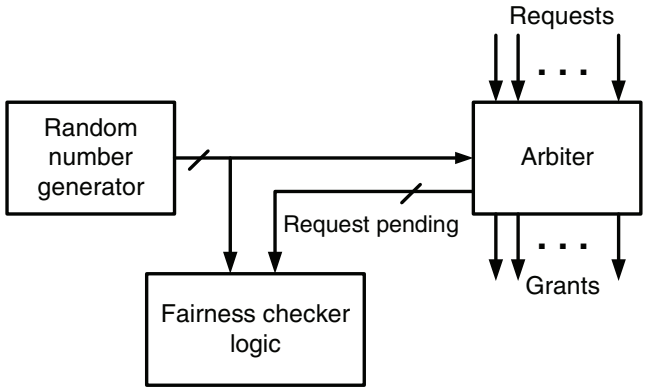


Fig. 2. Block schematic of the testbench with arbiter and fairness checker.

We use a 3 step process to quantify and verify the fairness properties of the random priority-based arbiter design under verification. In the first step, the request-to-grant delay bounds of the arbiter are determined in terms of CRS (instead of in cycles). In the second step, the lower and upper bound (in cycles) of the length of CRS generated by the random number generator (used to impose fairness by the arbiter, eg. LFSR) is determined. In the third step, the lower and upper bound values determined in step 2 are combined to compute the request-to-grant delay bounds (in cycles) of the arbitration logic as a whole. This is an accurate characterization of the performance of the arbiter, and can be used to verify that the design meets the specification. Each one of the above steps can potentially uncover a number of bugs in the logic design. A description of each of the 3 steps follows in the sections below.

A. Determining request-to-grant delays in CRS

In our first verification step, we quantify the upper bound on the request-to-grant delay, in terms of the number of CRSes in the random input number sequence, in the time interval from the issue of a request until it is granted. The idea is to prove a bounded liveness property – i.e. a request will be granted within a bounded period of time if the random number sequence meets certain fairness constraints.

This may be done by detecting the CRSes in the random number sequence and determining how many CRSes are needed to grant a request using a testbench consisting of a fairness checker logic, the arbiter and a set of non-deterministic input bits replacing the random number generator output signals as shown in Figure 2. Typically a subset of the outputs (via tap points on registers) of the random number generator is used by the arbiter; we replace these register outputs with non-deterministic input signals with the same timing characteristics. Note that we are not including the actual logic implementing the random number generator, such as an LFSR logic containing several registers, because in this step we are trying to prove a property of the arbiter independent of the characteristics of the particular random number generator

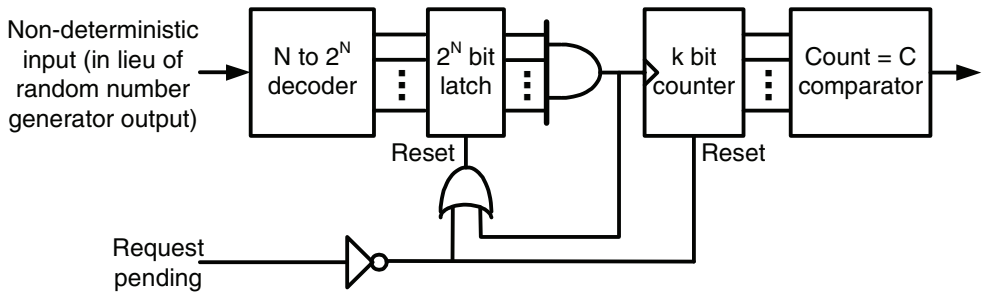


Fig. 3. Logic for checking fairness requirements of one input request of random priority arbiter.

driving the arbiter.

Figure 3 shows the fairness checker logic for one request. It uses a 2^N bit latch to keep track of the occurrences of the 2^N unique numbers in the random number sequence generated by a N bit non-deterministic input source (in lieu of the actual random number generator). A k bit counter is used to count the number of CRSes in the random number sequence while a request is pending (i.e., the time when a request is issued until the request is granted). The latch and the counter will be reset until a request becomes active. Whenever a CRS is detected in the input sequence, all the bits in the latch will be set, causing the output of the AND gate to become active. The k bit counter is incremented and the latch is reset after each time a CRS is detected. When a request is granted the *Request pending* signal becomes inactive. Thus, the k bit counter keeps track of the number of CRSes in the random number sequence while the request is pending. The size of the counter, k, may be selected based on the desired worst-case request-to-grant delay as required by the design specifications of the arbiter.

The k bit counter output values are compared with a constant value C using a comparator. Constant value C represents the request-to-grant delay in terms of the number of CRSes that the model checker will attempt to verify against in each one of the iterative proof steps. The initial value of constant C may be set to any value from 0 to $(2^k - 1)$ by the fairness checker logic. Regardless of the initial value of C selected, our verification method systematically searches the possible range of values of C in an iterative manner to find the largest value of C for which the following property holds for all possible sequences that can be generated by a random number generator:

Request pending $\wedge$ *(number_of_CRSes = C) = TRUE,*

where $C < (2^k - 1)$. The largest value of C can be determined in at most k verification runs using a binary search [17] approach for selecting the values of C in each run.

B. Determining the length of CRS

The second verification step accurately quantifies the upper and lower bounds of the length (in number of clock cycles) of CRSes generated by a specific implementation of random number generator, such as an LFSR, used by the arbiter. The basic idea is to nondeterministically sample the random number sequence for a fixed number of cycles L and varying L in each proof step until the sampled random number sequence has exactly one CRS. The maximum and minimum values of such a sampling window of length L gives the upper and lower bounds of the length of the CRSes, respectively, that can be generated by the random number generator.

A sampling window gating signal can be generated using a fixed width pulse generator comprising a counter that can count up to a fixed number of cycles (say L cycles) and triggered by a random start window signal as shown in Figure 4. Such a sampling window gating signal can be used to sample the random number sequence produced by a random number generator at random instances for a fixed time interval as shown in Figure 5. The sampled random number sequence (the output of 2-input AND gate in Figure 5) is monitored to detect whether the sequence contains one CRS in it using a logic similar to the fairness checking logic described in the last verification step (see section V-A).

Fairness checker logic allows using formal verification techniques to accurately determine the upper and lower bounds of the length of CRSes in the random number sequence by varying the fixed sampling window size L and proving that the fixed sampling window contains exactly one CRS. For a given fixed length L of the sampling window, value of the output signal "CRS found" of fairness checker logic shown in Figure 5 at the end of the sampling window may be used to prove that the random number generator may generate a CRS of length L. The property (absence of exactly *one* CRS in the random number generator output)

$$CRSfound = FALSE$$

holds only when $L < L_{min}$ and $L > L_{max}$, where L_{min} is the length of shortest CRS and L_{max} is the length of longest CRS. In other words, the length of the longest CRS can be determined by observing the values of $L > L_{min}$ for which the property holds. The length of the shortest CRS can be determined as the smallest value of L for which the property

$$CRSfound = TRUE$$

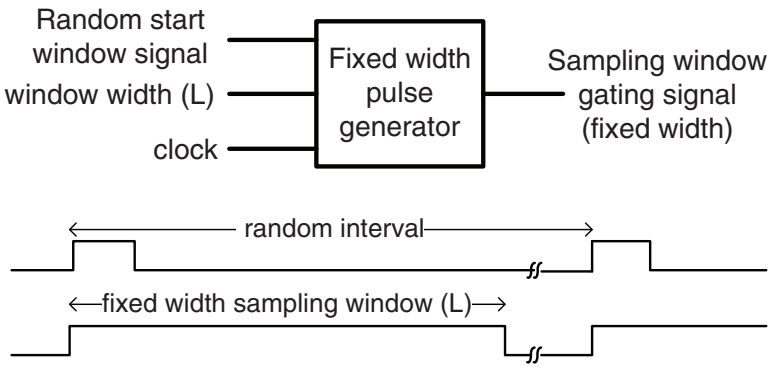


Fig. 4. Fixed width pulses with random periodicity.

holds for all possible window start cycles. The iterative process of determining L_{min} begins with the fairness checker logic initializing the sampling window gating signal to an initial fixed width $L = 2^N$. The value of L is incremented in each proof step until the above property holds.

The upper bound of the length of the CRS L_{max} can be determined in a similar way. The length of the sampling window is varied in each proof step such that the smallest value of the length of CRS $L > L_{min}$ for which the property

$$CRSfound = FALSE$$

holds for all possible random window start cycles. The value of L determined is $1 + L_{max}$.

C. Computing request-to-grant delay bounds

In the third step, the results of the first step and second step are combined to determine the worst case request-to-grant delay bounds in terms of the number of clock cycles of the entire arbitration logic, including the arbitration logic and the pseudo-random number generator logic driving the arbiter. In this third step, the upper and lower bounds of the length of the CRS computed in the second verification step may each be combined with the result (largest value of C) computed in the first step to obtain the worst-case request-to-grant delay bounds in cycles of the arbiter as:

(largest value of C) $\times$ (length of CRS in cycles)

The results obtained can be validated against the design specification to make sure that the arbiter conforms to the performance requirements, and no requests are starved for more than a fixed number of cycles.

Discussion The length of a CRS is determined by the last missing-random-number in a random number sequence regardless of the technique used for generating random numbers (such as LFSR logic). In the worst-case scenario, this last missing-random-number is the one that would cause a request to get a grant. Therefore, if the request arrives at the beginning of a CRS, it has to wait until such a last missing-random-number shows up to provide a grant, regardless of the way in which this last missing-random-number is delayed by the LFSR logic. This is the reason why the length of the longest CRS determines the worst-case request-to-grant delay.

Even though multiplying min/max length of a CRS with the number of CRSes can provide the range of request-to-grant delays corresponding to the best- and worst-case scenarios, it need not be an accurate representation of the actual worst-case request-to-grant delay bounds unless the LFSR can generate CRSes of the same (or similar) length back-to-back. It can be observed by analyzing the random number sequences generated by traditional LFSR designs that most of the time a series of back-to-back CRSes differ only by 1 cycle in length, followed by an abrupt change in the length of the CRS (see Figure 6), because of the way the LFSR is constructed with shift-registers and XOR gates in the feedback paths. Moreover, there are many possible permutations of random numbers ending with a specific missing-random-number - a phenomenon that is hard to avoid in traditional LFSR designs unless newer designs such as a *provably-fair random number generator* [18] is used.

However, for all practical purposes, the product of min/max lengths of a CRS and number of CRSes should provide us a fairly tight (+/- a few cycles) bound on the request-to-grant delay. If the design of the arbiter is such that any request can be starved for a fixed number (N) of CRSes, then in order to determine the *precise* request-to-grant delay bounds, we need to monitor the minimum and maximum length of N *contiguous* CRSes in the random number sequence in step 2. The CRS detection logic and method described in section V-B may still be used for determining the bounds of the length of N CRSes; the only difference is that we will be looking for the lower- and upper- bound of length of N contiguous CRSes instead of *one* CRS.

VI. Results

The formal verification method described has been used for verifying several random priority-based arbiters used in

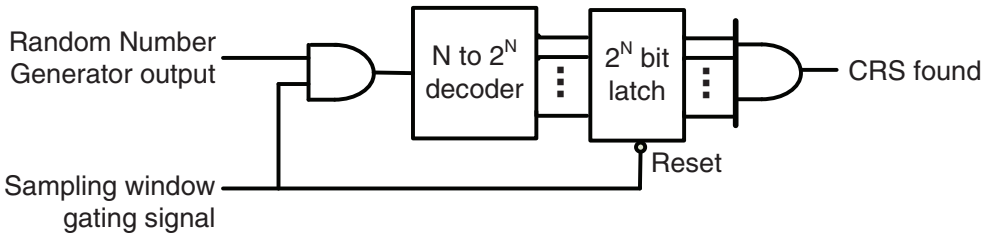


Fig. 5. Logic for checking the fairness of random number generator.

Design	Testbench Type	Traditional				CRS-based			
		Problem Size		Total Time	Peak Memory	Problem Size		Total Time	Peak Memory
		ANDs	Registers	(h:m:s)	(GB)	ANDs	Registers	(h:m:s)	(GB)
8to1ARB_RC	Any	2049	395	24:00:00	6.2	1738	333	0:0:27	0.071
	Multiple	3006	576	24:00:00	5.3	2625	513	0:01:43	0.091
	Bug	2880	554	24:00:00	16.9	2427	464	0:0:06	0.043
4to1ARB_SN	Any	2226	428	24:00:00	7.6	1284	302	0:2:40	0.144
	Multiple	3303	632	23:45:00	22.9	1971	455	0:7:03	0.168
10to1PBARB	Any	3852	770	17:33:07	18.1	2133	394	0:18:53	0.02
	Multiple	3855	770	24:00:00	15.2	3453	682	1:23:40	1.1

TABLE I
RUNTIMES AND MEMORY USAGE FOR DIFFERENT ARBITERS

```
12 | 735210000046
12 | 352100000467
12 | 521000004673
12 | 210000046735
12 | 100000467352
14 | 00000467352521
13 | 0000467352521
12 | 000467352521
11 | 00467352521
10 | 0467352521
10 | 4673525210
10 | 6735252104
10 | 7352521046
46 | 3525210463521421425252563142525210046314252567
45 | 525210463521421425252563142525210046314252567
44 | 25210463521421425252563142525210046314252567
43 | 5210463521421425252563142525210046314252567
42 | 2104635214214252525631425252100463142525 67
41 | 1046352142142525256314252521004631425256 7
40 | 0463521421425252563142525210046314252567
39 | 4635214214252525631425252100463142525 67
38 | 6352142142525256314252521004631425256 7
37 | 3521421425252563142525210046314252567
36 | 5214214252525631425252100463142525 67
35 | 2142142525256314252521004631425256 7
```

Fig. 6. A snippet of 3 bit random number sequence generated by a 16 bit LFSR showing one CRS per line. Each row depicts the length of the CRS followed by the actual CRS.

caches (eg. cache directory port arbitration logic) and on-chip interconnection network controllers (eg. command arbitration logic) of a commercial processor. Table I shows the experimental results on 3 of such industrial designs that use different types of random priority-based arbiters, and an LFSR to generate pseudo-random numbers. There are 3 types of testbenches: type *Any* is used to prove bounded liveness of any input request, whereas type *Multiple* is used for checking starvation of multiple requests. The *Bug* type is similar to the *Any*, except that a larger counter was used to generate a counter-example trace to help the designer fix a bug. All experiments were run on a 1.65 GHz POWER5+ processor with 384 GB memory using IBM's SixthSense (semi-) formal verification tool [14], a state-of-the-art industrial formal verification tool. We used a 2 bit counter ($k = 2$) in all the testbenches, except in the testbenches for 10to1_PBARB and 8to1ARB_RC (bug) in which we used a 4 bit counter. The requests are made non-deterministically to the arbiter with additional constraints on the arrival rate of the requests as per the design specifications. The problem size in terms of number of ANDs and registers, as reported by the SixthSense tool (since it uses AIGs [14] to represent the problem internally), shown is after initial cone-of-influence (COI) reduction and before unrolling the design.

The results are shown for both the proposed CRS-based and traditional bounded-liveness checking (counter-based) approaches. In the traditional approach, we included both the arbiter and the LFSR logic.

The request-to-grant delays determined were found to be in the range of 1 to 7 CRSes long. The bounds of the length of CRSes in the LFSR output used by the arbiters varied from a few tens of cycles to a few hundreds of cycles. For 8to1ARB_RC (bug) we were able to generate a short, yet illustrative, counter-example trace for the designer to identify the bug quickly. Note that using a counter in lieu of the LFSR would not have caught the bug, and in general should not be used as it does not analyze different permutations of random number sequences.

Each of the arbiter designs uses 16-stage LFSR logic. The size of the testbench used for computing the length of CRSes (as described in section V-B) is approximately 3000 ANDs

and 600 registers after initial COI reduction. Each run for determining the length of the CRS took anywhere from a few seconds to 6+ hrs, with an average overall runtime of 1.2 hrs to converge on the final value using a binary search approach.

We set a time limit of 24 hrs for each of the experiments. All the testbenches using the traditional bounded liveness approach were unsolved due to either exceeding the time, or memory limit (data structures exceeding bounds, eg. BDD node index). The total time reported for the latter ones is under 24 hrs. All the CRS based testbenches completed in a very short time using very little memory. It is clear from the experiments that the CRS based scheme significantly outperforms the traditional verification approach for random priority-based arbiters.

Our experience suggests that the proposed three step verification process helps to significantly reduce testbench complexity of random priority-based arbiters, thereby making them amenable for verification using formal verification. Moreover, the arbitration logic is verified in its entirety including the requirements imposed on the random priority-based fairness scheme.

In addition to verifying the correctness of the arbiter, the second verification step described in section V-B was used for tuning the pseudo-random number generators as well. Typically random priority-based arbiters are implemented using large LFSRs, and only a subset of the bits of the LFSR are used to assign priorities to the input requests. By monitoring the upper and lower bounds of the length of the CRS, one can easily choose the optimal tap points of an LFSR logic such that the logic/functional block using the arbiter meets the design specifications.

Furthermore, the request-to-grant delay bounds computed from the RTL-level design can be used for accurately modeling higher levels of hierarchy and larger systems (for example, the processor model) in which random priority-based arbiters are used. This in turn helps to more accurately estimate performance of the processor and systems, and contributes to overall performance verification.

It may be noted that though our main goal is to prove that none of the requests can be starved and to verify the request-to-grant delay specifications, the proposed verification method in effect proves that the arbiter cannot deadlock as well.

VII. Conclusions

A scheme to verify the fairness properties of pseudo-random number generators, and arbiters that use random priority-based arbitration is described. The presented approach uses a logic decomposition and formal verification based method to accurately characterize the worst- and best-case request-to-grant delay of such an arbiter inclusive of the fairness logic. The scheme first determines an upper bound on the request-to-grant delay of the arbiter in terms of the number of complete random sequences (CRS) independent of the pseudo-random number generator. It then separately determines, in terms of the number of clock cycles, an upper bound and a lower bound on the length of a complete random sequence in the random number sequence generated by a fairness logic used by the arbiter, i.e. the random number generator. Finally, the scheme determines a worst-case request-to-grant delay bounds of the arbiter system, in terms of the number of clock cycles, by combining the upper bound of the request-to-grant delay of the arbiter with the upper bound of the length of the CRS and the lower bound of the length of the CRS. The scheme has been successfully used to verify several random priority-based arbiters in the cache units and the on-chip interconnection network controllers.

Acknowledgments

Authors would like to thank Hari Mony for providing optimized SixthSense engine configurations and helpful discussions.

References

[1] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, 2004, ch. 18, pp. 350–362.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[3] L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Trans. Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[4] P. Horowitz and W. Hill, *The Art of Electronics*, 2nd ed. Cambridge University Press, 1989, pp. 665–667.

[5] A. Goel and W. R. Lee, "Formal Verification of an IBM CoreConnect TM Processor Local Bus Arbiter Core," in *Design Automation Conference (DAC)*, 2000, pp. 196–200.

[6] K. Wasaki, "A Formal Verification Case Study for IEEE-P.896 Bus Arbiter Using A Model Checking Tool," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 7, no. 3, pp. 184–191, 2007.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit verification using Symbolic Model Checking," in *Design Automation Conference (DAC)*, 1990, pp. 46–51.

[8] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] R. M. Gott, J. Baumgartner, P. Roessler, and S. I. Joe, "Functional formal verification on designs of pSeries microprocessors and communication subsystems," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 565–580, 2005.

[10] T. Le, T. Gloekler, and J. Baumgartner, "Formal Verification of a Pervasive Interconnect Bus System in a High-Performance Microprocessor," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 217–224.

[11] N. Dershowitz, D. Jayasimha, and S. Park, "Bounded Fairness," *Lecture Notes in Computer Science*, vol. 2772, pp. 304–317, 2004.

[12] A. Biere, C. Artho, and V. Schuppan, "Liveness Checking as Safety Checking," in *Proc. 7th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, ser. ENTCS, vol. 66, no. 2. Elsevier, 2002.

[13] V. Schuppan and A. Biere, "Liveness Checking as Safety Checking for Infinite State Spaces," in *Proc. 7th Intl. Workshop on Verification of Infinite-State Systems (INFINITY '05)*, ser. ENTCS, vol. 149, no. 1. Elsevier, 2006.

[14] H. Mony, J. Baumgartner, V. Paruthi, R. L. Kanzelman, and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2004, pp. 159–173.

[15] J. Baumgartner and H. Mony, "Scalable Liveness Checking via Property-Preserving Transformations," in *Design Automation and Test in Europe*, 2009.

[16] S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.

[17] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1997, vol. 3: Sorting and Searching.

[18] K. K. Kailas, B. C. Monwai, and V. Paruthi, "Method and Structure for Provably Fair Random Number Generator," U.S. Patent application 12/101,734, April 11, 2008, IBM docket No. YOR920080134US1.

Assume-Guarantee Validation for STE Properties within an SVA Environment

Zurab Khasidashvili, Gavriel Gavrielov
Intel Israel (74) Ltd.
Haifa 31015, Israel
{zurabk,ggavriel}@iil.intel.com

Tom Melham
Oxford University Computing Laboratory
Oxford, OX1 3QD, England
melham@comlab.ox.ac.uk

***Abstract*—Symbolic Trajectory Evaluation is an industrial-strength verification method, based on symbolic simulation and abstraction, that has been highly successful in data path verification, especially microprocessor execution units. These correctness results are typically obtained under certain assumptions about how the verified hardware block's inputs are driven, as well as assumptions about the values of these inputs. For correct overall operation, the hardware environment within which the verified block resides is expected to satisfy these assumptions.**

We describe a translation of these proof assumptions into System Verilog Assertions. These are then used as checkers in dynamic validation of the hardware environment within which blocks verified by Symbolic Trajectory Evaluation operate. The result is a pragmatic assume-guarantee method that increases the quality and confidence in verification results, requires little or no modification to the Symbolic Trajectory Evaluation proofs, and leverages pre-existing dynamic validation infrastructure.

I. Introduction

Symbolic Trajectory Evaluation (STE) is a model checking method based on symbolic simulation over a lattice of abstract state sets [1]. STE's combination of abstraction and algorithmic efficiency is especially suited to verification of datapaths and memories, and has been demonstrated on many hard industrial verification problems [2], [3], [4]. A notable success is the verification, using Intel's Forte system [5], of the entire execution cluster of the Intel Core 2 Duo and Core i7 microprocessors [6], [7]

As with most property verification, STE correctness results are usually conditional on various assumptions about the hardware environment within which the verified block is operating. Assumptions are made about how the inputs to the verified block are driven. And it is often assumed that the values presented on these inputs comply with certain constraints, sometimes quite complex ones. Validating these assumptions is part of the well-known *assume-guarantee* paradigm for compositional reasoning [8]. We separately verify a component under assumptions and show that the environment guarantees the assumptions hold.

In this paper, we describe an approach to assume-guarantee validation in which components are verified with STE and the environmental assumptions are translated into System Verilog Assertions (SVA) and checked with dynamic validation. The methodology has been applied on a large scale to STE verifications of a micro-controller unit, and of the microoperations in the execution cluster of a recently-designed Intel microprocessor.

This work encompasses methodology, theory, implementation, and experimental evaluation. Unrestricted STE is capable of expressing quite complex and possibly *implicit* environmental assumptions. Our methodological contribution is to describe a restricted, standardized form for encoding STE properties that enables us to translate their environmental assumptions—whether explicit or implicit—into equivalent, and efficiently checkable, SVA properties. Our theoretical contribution is a simple solution to capturing the semantics of certain uses of STE symbolic indexing variables in the SVA language, where there is no similar concept, and in arguing the soundness of the translation. We also describe an implementation that employs the reflection features of the Forte system's functional scripting language [9] to allow both STE verification and translation from a single source. Finally, we present experimental results from extensive use of the framework on the STE verification environment of a recently taped-out Intel microprocessor.

II. SVA and STE Background

In this section, we give just enough background on SVA and STE to enable the reader to follow the subsequent text and to appreciate the problem our translation addresses. A good tutorial introduction to SVA [10] can be found in [11]. A detailed introduction to STE is given in [1], and a full account of Intel's Forte environment appears in [5].

A. Basic SVA Notation

System Verilog Assertions provide a rich specification language for expressing assumptions and conditions on the values that appear on circuit nodes over time. The most basic form of assertions are expressions built up with operators over bits (circuit nodes) and bit-vectors (vectors of circuit nodes). These include negation (!), conjunction ($\&\&$), disjunction ($||$), implication ($<=$), equality ($==$), and bit-vector relations and functions such as comparison and addition. If E is an expression and k a non-negative integer, then '$\$\mathrm{past}(E,k)$' means the value E had k clock ticks ago. The expression '$\$\mathrm{stable}(E)$' says that the current value of E is the same as its value at the previous clock tick.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

An SVA *sequence* is an expression that describes a series of events over time. A delay operator is used to specify relative timing of events in the sequence. For example the sequence '$E_1 \; \#\#k \; E_2$' means that the expression E_1 holds and then, integer k clock ticks later, expression E_2 holds. Writing '$\#\#k \, E$' just means that the expression E holds k clock ticks from now. A repetition may be employed to say that an expression holds over some period of time. Writing '$E \, [*k]$' means that E holds for the next k clock ticks (including now).

Sequences can be combined using conjunction ('and') and disjunction ('or'). They can be inverted, to yield a *property*, with 'not'.

B. STE Verification and the Translation Problem

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.

Atomic propositions in A and C take the form '$P \rightarrow \mathrm{n}$ is 0' or '$P \rightarrow \mathrm{n}$ 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 n can have any value—including, for abstraction efficiency, the don't care value X. Antecedents and consequents are essentially just conjunctions (using ' and ') of these atomic propositions, possibly modified by the next-time temporal operator N.

The guards in a trajectory assertion are formulas of propositional logic over some Boolean variables. For each assignment of truth-values to the variables, the assertion collapses into a property checkable by three-valued simulation, with the don't care value X on all circuit nodes not forced to 0 or 1 by the antecedent (or the circuit). Given a trajectory assertion, STE *simultaneously* computes all these three-valued simulations, and checks the results against the consequent C.

This arrangement gives STE a native capability for partitioned Boolean abstraction that, by the method known as *symbolic indexing* [12], can be very effective on large but 'semantically regular' datapaths. A full discussion of symbolic indexing and its automation can be found in [13] and [14].

This machinery can encode sophisticated abstraction schemes with complex implicit environmental assumptions. In the work of this paper, however, our methodology imposes constraints on how STE properties are written in order to make their assumptions easily translated into SVA. In the simplest case, we deal with STE sub-formulas of the form

$$x \rightarrow \mathrm{n} \text{ is } 1 \;\; \text{and} \;\; \neg x \rightarrow \mathrm{n} \text{ is } 0$$

where x is a Boolean variable that appears nowhere else in the formula and n is a circuit input node. This establishes a one-to-one correspondence between x and n—the circuit input n is 'driven by' the symbolic Boolean variable x. We abbreviate this by writing 'n is x', and more generally allow one to write 'n is E' for any propositional formula E. If a unique, unconstrained Boolean variable is associated with each circuit input in this way, then STE verification is essentially just symbolic simulation.

Environmental assumptions can be added to this scheme by using the *parametric encoding* [15] of Boolean constraints. For example, if the antecedent of a two-input device is

$$\mathrm{a} \text{ is } x \;\; \text{and} \;\; \mathrm{b} \text{ is } y$$

we can make the assumption that at least one input is high by using the parametric technique to verify this property under the assumption $x \vee y$. Using this method, any formula that constrains the variables in an STE antecedent can be taken as an environmental assumption.

In this simple example there is a one-to-one correspondence between circuit nodes and variables, so it is easy to re-express the assumption as an SVA expression over circuit nodes. We just substitute nodes for corresponding variables, giving the SVA expression $\mathrm{a} \,||\, \mathrm{b}$. The propositional disjunction, $\vee$, becomes SVA disjunction, $||$.

We wish, however, to support more subtle use of STE than this—including limited forms of symbolic indexing for abstraction efficiency. We commonly find, for example, STE antecedents with guarded sub-formulas equivalent to

$$P \rightarrow \mathrm{a} \text{ is } x \;\; \text{and} \;\; Q \rightarrow \mathrm{b} \text{ is } x \tag{1}$$

Now there isn't a direct correspondence between variables and nodes. Sometimes the variable x is associated with node a, and sometimes with node b—and, if P and Q overlap, sometimes with both. There is no obvious node name to replace occurrences in environmental assumptions of the variable x (or, indeed, in the guards of *other* sub-formulas). Moreover, this antecedent makes an implicit assumption about circuit nodes, namely that the value on a is the same as the value on b when P and Q both hold.

A general method to untangle completely unrestricted STE antecedents and environmental assumptions into equivalent SVA properties would be very complex. Our work therefore places methodological restrictions on the form in which STE antecedents are written. Part of our contribution—explained in the sections that follow—is to propose a form that scales to the large verifications done at Intel, is natural for engineers to write, and allows symbolic indexing—while still admitting of a fairly intuitive translation into SVA.

C. The 5-tuple Representation of Antecedents

The Forte implementation of STE is embedded in a functional programming language, *reFLect* [9], similar to ML. STE antecedents are represented by lists of *5-tuples* of the form

$$(guard, node, value, start, end)$$

where *guard* and *value* are formulas of propositional logic (represented as BDDs), *node* is a node name (a string), and *start* and *end* are non-negative integers. The meaning is that if *guard* holds, then *node* has *value* during the

time period from simulation cycle $start$ up to but excluding simulation cycle end. So, for example, the 5-tuple list $[(P, \mathrm{a}, x, 0, 1), (Q, \mathrm{b}, x, 0, 1)]$ is the machine representation of the STE antecedent (1) above. Our methodological restrictions and translation are defined over these 5-tuple lists, which can represent any STE antecedent formula.

III. STE Proof Environment

Our target application is verification of micro-operation (μop) execution by, for example, the execution (EXE) cluster of a contemporary X86 microprocessor. We will call the region of the full chip under verification the *STE unit*.

There are typically several thousand different μops, executed in several sub-units of the STE unit and at several ports. To organize and share STE proof code, μops are divided into groups. The μops in a group are normally executed in the same sub-unit and on the same port. Their STE antecedents are therefore very similar. There can be several μop groups for the same port.

For each UOP group, the STE antecedent and any accompanying environmental constraints are divided into the following components:

a) Timed assumptions: This is an STE 5-tuple list that describes how inputs to the the STE unit are driven (over time) when a μop is executed. Timed assumptions must be satisfied by the post-reboot behaviour of the full chip—their validity as STE assumptions is expected to be guaranteed by the hardware that drives the inputs of the STE unit.

b) Timed restrictions: This is an STE 5-tuple list used to ignore uninteresting full chip (and STE unit) behaviour over time—behaviour we do not wish to verify in STE. For example, if μop execution is interrupted by a reset or if certain input validity signals are not asserted when μop execution starts, then we don't want STE to check correctness of the output. Timed restrictions often set STE unit inputs to constants. For example, we might make an input F to ignore the STE unit's behaviour for an incompletely implemented feature that is invoked when that input is T. Such restrictions are either temporary and will be removed at a later stage of the design, or they will be used for case-splitting all possible legal behaviours of the STE unit.

c) Global assumptions: These are conditions on Boolean variables that, in the context of the STE antecedent, characterize relationships between values on the STE unit's input that should be satisfied by the post-reboot behaviour of the full chip whenever a valid μop is executed. Some global assumptions may apply to only part of a μop group. Some global assumptions characterize expected microcode behaviour. As with timed assumptions, the validity of global assumptions is expected to be guaranteed by the hardware that drives the inputs of the STE unit.

d) Global restrictions: These are constraints on Boolean variables that focus the proof on some interesting scenario of the STE unit. This is mainly useful for case splitting. For example, one might want to restrict the verification to a particular group of μops. The condition that the executing μop belongs to that group will be a global restriction on Boolean variables representing the opcode of the μops. As with timed restrictions, global restrictions may be used to ignore specific behaviours of the STE unit temporarily.

The distinction between *assumptions* and *restrictions* is pragmatic—assumptions are expected to be met by the operating environment (i.e. are the subject of our assume-guarantee reasoning) and restrictions are not. The distinction between *timed* and *global* is syntactic—timed constraints come from 5-tuples, and global constraints come from expressions over Boolean variables.

In STE verification, the timed restrictions and assumptions are taken together form the antecedent. The conjunction of the global restrictions and assumptions constitutes a Boolean constraint on STE variables that is taken as an assumption using the parametric technique [15]. Thus the distinction between assumptions and restrictions is irrelevant for the STE proofs. On the other hand, the STE consequent is irrelevant for the assume-guarantee method supported by our translation.

IV. STE to SVA Translation

From the description above, it follows that the restrictions can be violated by legal, full chip simulation traces. But for the parts of traces that do violate the restrictions, the STE unit's output behaviour is (by definition) irrelevant, and STE passes vacuously. In our assume-guarantee method, therefore, the generated SVA must ignore the parts of full chip simulation traces where any restriction is violated.

This is done by using the SVA property generated from the restrictions (timed and global) as a *trigger* for all the *checker* properties generated from the assumptions (timed and global). Every System Verilog Assertion we generate has the following implicational form:

$$\text{not } trigger \text{ or } checker$$

In our methodology, the trigger is the same for every generated SVA for a group of μops. One or more checker properties are built from each tuple in the timed assumptions and from each global assumption. We observe that the translation preserves abstraction, in the sense that whenever the STE proof environment makes no assumption about the value on a circuit node (i.e. that node gets value X in the STE simulation), the generated SVA makes no constraint on the value of that node.

A. Preprocessing

Let A be the conjunction of all global assumptions and restrictions. As a preprocessing step, we replace every tuple (g, n, v, s, e) in the timed restrictions and timed assumptions with (T, s, v, s, e) if $A \Rightarrow g$ is a tautology, and omit the tuple if $A \wedge g$ is unsatisfiable. This is sound because the global restrictions are part of the SVA trigger, and any violation of the global assumptions will be caught by the assertions obtained by translating them into SVA.

Similarly, let R be the conjunction of all global restrictions. As a preprocessing step, we inspect each global assumption g. If $R \Rightarrow g$ is a tautology, we do not generate SVA for g; if

$R \wedge g$ is unsatisfiable, we replace g with F. Such an assertion would fail every time the simulation check is triggered, so the STE-to-SVA translation immediately reports such global assumptions.

B. Translation of Ground Tuples

A *ground tuple* is a 5-tuple (T, n, v, s, e) in which the guard is true and v, the value on node n, is a Boolean constant, T or F. If v is T, then the tuple is translated to the SVA sequence $\#\#s\, n\, [*e-s]$: after a delay of s, node n is high for $e-s$ time units. Similarly, if v is F, the SVA sequence is $\#\#s\, !n\, [*e-s]$.

C. Handling STE Boolean Variables

The guard of any non-ground tuple in the timed restrictions and timed assumptions will be a non-constant expression over one or more Boolean variables. The value component may be T, F, or an expression over some Boolean variables. Any of these variables may occur in the propositional constraints that constitute the global assumptions and restrictions.

As suggested in section II, our translation to SVA works by finding, for each Boolean variable used in the STE verification, a group of circuit nodes that can serve as proxies, or representatives, for that variable in the corresponding SVA expressions. Global assumptions and restrictions are re-expressed as SVA constraints on those circuit nodes. Likewise, explicit or implicit relationships among values in the circuit imposed using variables in STE are re-expressed as SVA properties over circuit nodes.

To make this work, we impose some methodological constraints on the way in which STE antecedents are written. We say that a variable x *immediately depends* on y if there is a tuple with x in its value and y in its guard. We introduce a dependency relation between variables as the transitive closure of immediate dependency. Using this concept, we impose the following restrictions on the usage of variables in antecedents. Within each group of μops, the following hold:

- Each Boolean variable, whether in the antecedent or in the global assumptions and restrictions, 'drives' at least one input node of the STE unit. That is, each variable is the entire value expression of at least one timed assumption or timed restriction tuple.
- the dependency relation between variables is a strict partial order. Variable dependency defines a DAG.

The idea is to ensure that for every Boolean variable in STE there is a circuit node that can be used as an SVA expression in the translation that denotes the value of that variable. In fact the conditions are a little more complex than the above, because in STE Boolean variables can both drive nodes and be used in expressions conditionally, depending on guards and the global restrictions and assumptions. The general rule is that whenever a variable is used, it must also be 'defined', in the sense of driving a specific circuit node.

Both these conditions are methodological—STE works without them, but then it is not amenable to the assume-guarantee reasoning we're trying to support. One way to deal with variables that do not drive circuit nodes would be to perform a case splitting on them. But this is not practical because each case split can double the amount of generated SVA. The acyclicity restriction ensures there is at least one circuit node that can serve as a representative for each variable.

D. Finding Circuit Node Representatives for Variables

The fundamental element of our algorithm is translation of propositional expressions over Boolean variables into equivalent SVA expressions over circuit nodes. The mapping of Boolean and bit-vector operations—negation, conjunction, binary addition, and so on—to corresponding SVA operations is straightforward. Variable translation is handled as follows.

Let x be any variable that occurs, on its own, as the value expression of any tuple in the timed restrictions or timed assumptions. There may be several such tuples. For any x, we partition the set of all such tuples into subsets $\mathcal{T}_x^g$, where g is a distinguished *maximal guard* among all the guards of tuples in the subset. More precisely, g is logically implied by each of the guards of the tuples in $\mathcal{T}_x^g$. (If there is more than one such maximal guard, we make an arbitrary choice). Moreover, we suppose this partitioning is such as to ensure that if we have any two subsets $\mathcal{T}_x^{g_1}$ and $\mathcal{T}_x^{g_2}$, then $g_1 \wedge g_2$ is unsatisfiable.

The idea is that when g holds, the node components of the tuples in $\mathcal{T}_x^g$ are all candidate representatives for the variable x in our translation. We choose one such candidate as follows. Let $s(x, g)$ be the earliest start time of the tuples with a maximal guard in $\mathcal{T}_x^g$, and let $n(x, g)$ be the node component of the tuple in $\mathcal{T}_x^g$ with this earliest start time. (In case of ties, we make an arbitrary choice.) Let f (for *future*) be any integer greater than or equal to the end time of every tuple in $\mathcal{T}_x^g$ for all relevant guards g. We define a standard SVA 'name' for the value of variable x when g holds as follows:

$$node(x, g, f) = \$\mathrm{past}(n(x, g), f - s(x, g))$$

Suppose, for example, that

$$\mathcal{T}_x^P = \{(P, \mathrm{a}, x, 2, 5), (Q, \mathrm{b}, x, 1, 4)\}$$

where $Q \Rightarrow P$. Then $node(x, P, f)$ is the SVA expression '$\$\mathrm{past}(\mathrm{a}, f-2)$', for any $f \geq 4$. The function of f is to relativize time points with respect to a reference point of time somewhere beyond the end of the relevant segment of the STE simulation run for these tuples. We discuss this further in section IV-H.

Given a Boolean expression P, we compute a family of SVA translation instances as follows. Let $\mathcal{V}_P = \{x_1, \ldots, x_k\}$ be the set of all variables in P, together with all the variables they depend on, according to our dependency relation. For each variable x_i, where $1 \leq i \leq k$, we choose a maximal guard g_i that appears in one of the tuples in which x_i is the value expression. Let f be any integer greater than or equal to the end time of every tuple in every $\mathcal{T}_x^g$, for $x \in \mathcal{V}_P$ and all relevant guards g. We define a mapping from variables to SVA expressions:

$$\theta_f = \{x_i \mapsto node(x_i, g_i, f) \mid 0 \leq i \leq k\}$$

Thus θ_f is determined by choosing a relevant maximal guard for each variable that P depends on. For any Boolean formula Q, we write $Q\theta_f$ to denote the translation of Q into an SVA expression, with Boolean variables mapped to SVA sub-expressions using θ_f and with the obvious replacement of Boolean operations by SVA operations.

For a given Boolean expression P and a given θ_f defined relative to P, we define a translation instance of P to an SVA expression as follows:

$$exp(P, \theta_f) = ((g_1\theta_f \mathbin{\&\&} \cdots \mathbin{\&\&} g_k\theta_f) <= (P\theta_f))$$

where $g_1, \ldots g_k$ are the guards chosen for the variables $x_1, \ldots, x_k$ in defining θ_f. The methodological restriction that variable dependency forms a DAG ensures that the domain of θ_f covers all the variables that occur in $g_1, \ldots, g_k$. We then define an SVA *sequence* for P,

$$seq(P, \theta_f) = \#\#f\ exp(P, \theta_f),$$

by applying a top-level delay operation '$\#\#f$'.

1) Conflicting Mappings: There may be many translations of a given propositional formula P into an SVA sequence, one for each choice of guards for the variables that P depends on—i.e. one for each mapping θ_f we can construct. Our overall translation needs to cover all relevant cases, but some irrelevant cases may be eliminated as follows.

The formula P we are translating can be the guard or value expression of a tuple, a global restriction, or a global assumption. When P is a guard or a value component, we use the conjunction $R \wedge A$ of global restrictions and assumptions to filter out mappings that are ruled out by them ('conflicting mappings'). If, for some selection of guards $g_1, \ldots, g_k$, we find that $g_1 \wedge \ldots \wedge g_k \wedge R \wedge A$ is unsatisfiable, then we do not include the corresponding θ_f among the mappings we use when computing relevant translation instances of P. Similarly, when P is a global assumption, then we use the conjunction of all global restrictions, instead of $R \wedge A$, to eliminate impossible mappings.

We denote the set of all non-conflicting mappings for a Boolean formula P by $maps(P)$, and we define

$$Exp(P, f) = \&\&_{\theta_f \in maps(P)}\, exp(P, \theta_f)$$
$$Seq(P, f)\ = \#\#f\ Exp(P, f)$$

for any sufficiently large value of f.

E. Generating Equality Expressions for Variables

STE antecedents commonly drive several different circuit nodes with the same variable. This encodes an implicit assumption that our translation must capture, namely that the same value appears on all the circuit nodes that are driven by the same variable.

For a pair of tuples (g_1, n_1, x, s_1, e_1) and (g_2, n_2, x, s_2, e_2), we define an *equality constraint* as follows:

$$\#\#f \begin{pmatrix} Exp(g_1 \wedge g_2, f) \\ \quad <= \\ \$\mathrm{past}(n_1, f{-}e_1) == \$\mathrm{past}(n_2, f{-}e_2) \end{pmatrix}$$

Note that here we need to use a reference time f and 'past' expressions, since we cannot write $\#\#e_1\, n_1 == \#\#e_2\, n_2$ in SVA syntax.

Of course we simplify $g_1 \wedge g_2$ when one of g_1 or g_2 implies the other. And when $g_1 \wedge g_2$ is unsatisfiable, we optimize by not generating an equality constraint for this pair.

1) Equality within timed restrictions: We divide the restrictions that have a variable as the value expression into groups, according to the variable. For each group, with defining variable x, we generate equality constraints as follows. Partition the group into maximal subsets $\mathcal{T}_x^{g_i}$, with $1 \leq i \leq k$. For each set $\mathcal{T}_x^{g_i}$, generate an equality constraint between the tuple with guard g_i and every other tuple in the set. Then generate an equality constraint between the tuple in $\mathcal{T}_x^{g_i}$ with guard g_i and the tuple in $\mathcal{T}_x^{g_j}$ with guard g_j, for $1 \leq i < j \leq k$. The SVA sequence conjunction of all these constraints is used as a conjunct in the trigger.

2) Equality within timed assumptions: Similarly, we divide the timed assumptions that have a variable as the value expression into groups by variable. Generate equality constraints that link the node values of the tuples within each group, using a partitioning by maximal guards as for the timed restrictions. Each of these constraints serves as a checker property in the SVA generated from the timed assumptions.

3) Equality relating timed restrictions and timed assumptions: Finally, we generate equality constraints for any variable x that occurs as the value expression in the timed restrictions and in the timed assumptions. The constraints are generated according to the partitionings introduced above. For each subset $\mathcal{T}_x^{g_a}$ of the timed assumptions and each subset $\mathcal{T}_x^{g_r}$ of the timed restrictions, generate an equality constraint between the tuple in $\mathcal{T}_x^{g_a}$ with guard g_a and the tuple in $\mathcal{T}_x^{g_r}$ with guard g_r. The conjunction of all such constraints is used as a conjunct in the trigger.

F. Generating SVA for Global Assumptions and Restrictions

Let $R_1, \ldots, R_n$ be all the global restrictions. For each $1 \leq i \leq n$, we generate the SVA sequence $Seq(R_i, f)$. Note that this sequence covers all possible mappings of variables to nodes that arise from choosing combinations of guards for the variables R_i depends on. We then include the SVA sequence conjunction of all these sequences as a conjunct of the trigger.

Let $A_1, \ldots, A_m$ be all the global assumptions. For each A_i with $1 \leq i \leq m$, and for each θ_f in $maps(A_i)$, we generate a corresponding SVA sequence $seq(A_i, \theta_f)$ as a checker expression in the SVA generated by our translation.

G. SVA Properties for each Antecedent Tuple

For any tuple (g, n, v, s, e), we define its *relevant variables* to be all variables that occur in g or v, plus the variables on on which these depend. For each mapping θ_f covering all the relevant variables of a given tuple, we generate one or more SVA expressions as follows.

If the tuple has the form (g, n, T, s, e), i.e. the value component is constant T, we generate the SVA property

$$\mathrm{not}(Seq(g, f)) \text{ or } (\#\#s(n)\,[*e{-}s])$$

Similarly, if the tuple has the form (g, n, F, s, e) we generate

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } (\#\#s(!n)\,[*e{-}s])$$

These simply say, in SVA, that the nodes mentioned in these tuples have the constant values given by the STE antecedent within the stated windows of time.

For every tuple (g, n, x, s, e) with a variable x as its value component, we generate the SVA property

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } (\#\#s{+}1(\$\mathrm{stable}(n))\,[*e{-}s{-}1])$$

This says only that the value on node n is stable over the period during which it is driven by the variable x in STE; linkage with ocurrences of x in other tuples is handled by the equality constraints. We omit the generation of this stability condition if $e - s \leq 1$.

For tuples (g, n, v, s, e) where the value component v is not a constant or a variable, we generate the SVA property

$$\mathrm{not}(\mathit{Seq}(g, f)) \text{ or } \#\#f(\$\mathrm{past}(n, f - s) == \mathit{vname})$$

where vname is a fresh SVA identifier defined to be equal to $\mathit{Exp}(v, f)$.

The SVA properties generated using these rules from the tuples in timed restrictions all become conjuncts of the trigger. Each SVA property generated from a timed assumption tuple becomes a checker expression.

H. Summary Overview of the Algorithm

As already mentioned, the top-level SVA properties generated for assume-guarantee validation by our method are all of the form '$\mathrm{not}\ \mathit{trigger}\ \mathrm{or}\ \mathit{checker}$'. There is one global $\mathit{trigger}$ property per μop group, the conjunction of:

- all SVA properties generated from the tuples in the timed restrictions (section IV-G),
- the equality constraints for variables within the timed restrictions (section IV-E1),
- the equality constraints relating timed restrictions and timed assumptions (section IV-E3), and
- SVA sequences for the global restrictions (section IV-F).

For each of the conjuncts in this trigger property, our implementation chooses an appropriate value for the reference time 'f' of our translation. The efficiency of checking the resulting SVA in simulation depends critically on the depth of simulation required by the 'past' operators in our properties. Our implementation therefore includes a complex algorithm that aims to minimise f for each conjunct in the trigger.

In SVA syntax, the trigger property is written as a conjunction of the above constraints—is illustrated by this example:

$$\begin{array}{l}\mathit{property\ trigger};\\ @(\text{`}\mathit{SYS\_CLK})\\ \quad((\#\#10(\$\mathrm{stable}(\mathrm{opcode}[2:0]))\,[*1])\ \mathrm{and}\\ \quad\ (\#\#5(!\mathrm{reset})\,[*12])\ \mathrm{and}\\ \quad\ (\#\#7(\mathrm{uop\_valid})\,[*2])\ \mathrm{and}\\ \quad\ \#\#10(\mathrm{uop\_group\_condition}))\\ \mathit{endproperty}\end{array}$$

This aligns the time units within our conjunct sequences with, $\mathit{SYS\_CLK}$, which is the reference clock in the system.

The individual $\mathit{checker}$ properties comprise:

- each SVA property generated from the tuples in the timed assumptions (section IV-G),
- the equality expressions for variables within the timed assumptions, (section IV-E2),
- the SVA sequence for each global assumption (section IV-F).

Again, the reference time f for translation of each SVA property is minimised for simulation efficiency.

For each checker, an SVA assertion is defined that says the checker must hold triggered. Suppose the reference time f is 10. Then a typical example is the following stability property for the tuple $(\mathrm{T}, \mathrm{output}, \mathit{variable}, 4, 9)$:

$$\begin{array}{l}\mathit{output\text{-}stability\text{-}assertion} : \mathit{assert\ property}\\ \mathrm{not}\ \mathit{trigger}\ \mathrm{or}\ \#\#10(\$\mathrm{stable}(\$\mathrm{past}(\mathrm{output}, 5))\,[*4]\end{array}$$

A pragmatic optimization, in the default flow, is that we do not generate SVA corresponding to STE tuples that drive clocks. (In STE verifications, clocks need to be driven explicitly by the antecedent.) If the clocks do not behave correctly, this will be caught by other validation activities.

V. Implementation

Our translation method is implemented within Intel's Forte system [5]. Forte is essentially a programming environment based around the *reFLect* functional programming language [9], and large-scale STE verification efforts are to a great degree a programming (or scripting) activity in nature. STE model checking is invoked through *reFLect* library function calls, and trajectory assertions for verification (lists of 5-tuples) are generated by writing functional programs that compute them. Our translation is also implemented as a *reFLect* functional program that computes SVA texts from lists of 5-tuples together with the Boolean expressions stating global restrictions and assumptions.

A. Usage of Reflection

The STE proof environment contains calls to functional program code that generate many Boolean formulas for each μop group being verified—guards, value expressions, global assumptions, and global restrictions. This code is user-defined and of arbitrary complexity, and therefore unsuitable as the source for our translation into SVA. On the other hand, the resulting formulas are represented in *reFLect*, and passed to the STE model checking engine, as BDDs (or, optionally, a form of AIGs [16]). This makes them too low-level for translation into SVA—once we have evaluated down to a BDD, much useful structure is lost and compact translation is difficult.

Our implementation solves this problem by exploiting the reflection features of *reFLect* to 'intercept' the evaluation of function calls that generate Boolean formulas at a stage suitable for translation into SVA. The *reFLect* language includes a primitive datatype, *term*, whose elements are the abstract syntax trees of *reFLect* programs themselves. Functions can

take terms as arguments, analyse their structure, and return terms as results, in any programmable way. We exploit this to replace the BDDs that occur in 5-tuples by terms that represent syntax of the *reFLect* function calls that generate these BDDs, at a level of elaboration suitable for translation to SVA.

Suppose there is a *reFLect* function, xor say, that normally computes a BDD, but which we wish our translation to see. We overload the *reFLect* function identifier 'xor' with an alternative version that takes terms, rather than BDDs, as arguments and produces the *reFLect* syntax tree consisting of an application of the xor function to these terms. This alternative xor builds an exclusive-or *expression* rather than a BDD. We do this for all functions we want to stop evaluation at—i.e. all those functions that we wish to form the vocabulary for the source language of our translation to SVA. The arbitrarily complex STE proof environment built on these primitives will then, through overloading, have two interpretations—one that computes 5-tuples with BDDs, for STE, and one that computes 5-tuples with terms for translation into SVA.

By this method we are able to have a single Forte source for both running STE and generating SVA. Verification engineers need to make very few changes to their existing STE environment to enable this new flow. Moreover we ensure that the Boolean expressions we translate are at a suitable abstraction level for to produce compact, efficiently checkable, and human-readable SVA.

B. Vectorization

Primitive STE antecedents express everything in terms of individual bits, and it is difficult to ensure that the STE proof environment keeps bit-vectors intact through to the level at which translation to SVA begins. As a second step in preprocessing, therefore, we perform *vectorization*—grouping of the nodes of STE tuples that belong to the same bit-vector. This enables much more compact and efficient SVA to be generated, with vector operations instead of bit-level ones. Tuples whose nodes belong to the same vector (we can tell from their names) and whose guards are the same are merged into a vector tuple $(g, \vec{n}, \vec{v}, s, e)$. We do this whenever all the bits of $\vec{v}$ are variables whose names indicate they belong together, or all the bits of $\vec{v}$ are T or F. Constant vectors are translated to hexadecimal numbers in SVA.

VI. Experimental Results

Our assume-guarantee mechanism has been used at Intel on two major examples, a micro-controller unit, and μop execution in the execution cluster of a recently-designed processor. We give some results from the second of these.

SVA properties were generated and checked by dynamic validation for every μop group (except multiplication and division) in the STE proof environment of the execution cluster. A total of 36 μop groups were covered, comprising 1,035 μops in total. The number of μops per group ranged from 1 to 111, with an average of around 29. A total of 3,616 SVA checker properties were generated, of which 3,061 were from global assumptions, 471 were from constant assignment

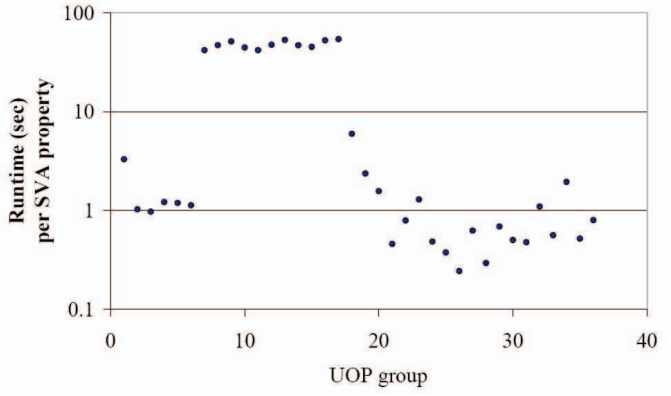


Fig. 1. Runtime per SVA Property Generated for EXE Cluster μop groups.

tuples, and 84 were from equality constraints. There were no equality constraints relating timed restrictions and timed assumptions, and no non-trivial stability constraints—these parts of our translation were tested in other exercises.

VCS, a 3-valued simulator from Synopsys, was run on all 3,616 SVA checker properties in one VCS session with 173 cluster level tests. The runtime was approximately 54.5 hours. Running the same tests on the execution cluster without any SVA checks (just computing the waveforms of all nodes) takes around 27 hours. So the overhead of checking the SVA generated by our method was a factor of around 2.

The data in figure 1 show the runtime for our translation (in seconds, on a log scale) per SVA property generated, for each group. The average runtime is 15.4 seconds. The high runtimes for some groups is due to the pre-processing steps of our algorithms, which require intensive BDD computations. This high runtime is compensated for by a reduction in SVA properties generated, and so in VCS simulation time. For example, in one of the groups taking over 500 seconds, the number of timed restrictions was reduced by pre-processing from 138 to 28, and number of timed assumptions from 86,970 to 1,991. For another group, the numbers are 92 to 24 for timed restrictions and 86,790 to 1,969 for timed assumptions. By contrast, for a μop group with runtime 11 seconds, the numbers are 92 to 24 and 4,078 to 2,017.

Without vectorization, it is estimated we would have at least two orders of magnitude more assertions, with very large combinational expressions, and it would not be practical to check all the SVA properties in VCS. More importantly, we found a huge benefit in keeping symbolic the bit-vector that codes for the instruction field of the μops in each group. In the STE proofs, the μop code sometimes needs to be made concrete for STE capacity reasons—essentially these proofs case split over the different kinds of μops in each group. In our translation, however, we code the μop instruction with Boolean variables, and translate the group as a unit. (Groups 21–36 are too complex to handle in this way, and were run with explicit μops.) A user who run all the sessions without this symbolic UOP feature generated 100 times more properties,

beyond the capacity of VCS.

A. STE Environment Violations and Bugs Found

1) Unused variables: Our methodology requires every variable in the global assumptions to drive at least one circuit node—our tool reports an error if this is violated. In our experiments, there have been tens of such cases. Most of the time such variables were redundant, and the STE proofs still ran after cleaning up the global assumptions and restrictions to eliminate them. In rare cases, debugging unused variables revealed true environment violations, and after correcting them the STE proofs failed. In such cases, debugging unused variables lead to eliminating false positives in the STE proofs.

2) Assertion failures: When an assertion generated by our translation fails in dynamic validation, there are two possible reasons. First, the assertion may simply be too strong, and eliminating the corresponding assumption (implicit or explicit) from the STE environment simply strengthens the the STE proofs. Second, the assertion may catch a real environment violation—a *potential* false positive in the STE proofs. Running the generated SVA for the 173 cluster level tests mentioned above revealed tens of wrong or redundant assumptions in the STE environment.

3) Bugs found: After the cleanup stage, the 3,616 SVA were run in VCS on 1,100 core level tests. Two bugs were discovered in the design as a result of this activity. These bugs were found in the interaction between the microcode and the execution cluster, rather than in the execution cluster itself. This is to be expected, because the assume-guarantee activity supported by our work starts at a relatively late stage of validation, when the bugs that have already been found using STE have been fixed.

VII. Conclusion

There is, of course, a vast literature on assume-guarantee reasoning in formal verification. In this paper, we have described a pragmatic method for checking the assumptions of STE proofs by dynamic validation in an SVA environment, rather than checking them by proof. The method has reasonable computational overhead, doesn't require users to make substantial modifications to their existing STE proofs, and has shown useful benefits in experimental usage. We have found that our methodological constraints are not burdensome in practice, and that we can handle all the forms of environmental specification that arise in industrial examples.

The task of verifying the quality of STE antecedents has also been addressed at Intel by translation into checkers (in a different language) with less aggressive vectorization [7]. This translation imposes a different structuring methodology on the STE environment that allows more flexible symbolic indexing, but also rules out implicit equality constraints coded by tuples. The assertions generated appear larger and significantly more numerous than with our method. On the other hand, this method can handle more complex STE proof environments.

Recently we have also used the SVA produced by our translation to generate stuck-at-tests for execution cluster outputs. Here, the SVA properties were used as assumptions. This experimental application aims to leverage the effort put into creating an STE proof environment for post-silicon validation.

Acknowledgments

The authors thank Yulik Feldman and Dmitry Korchemny for helpful advice about SVA, and Zenya Belfer for assistance with running the VCS simulator. Shachar Finkelstein was largely responsible for deploying STE2SVA and provided us with benchmark data. Roope Kaivola kindly provided helpful comments on a draft of the paper.

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, 1997, pp. 167–172.

[3] 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.

[4] R. Kaivola and K. R. Kohatsu, "Proof engineering in the large: formal verification of Pentium® 4 floating-point divider," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 323–334, 2003.

[5] 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.

[6] A. Flaisher, A. Gluska, and E. Singerman, "Case study: Integrating FV and DV in the verification of the Intel® Core™2 Duo microprocessor," in *Formal Methods in Computer Aided Design: FMCAD 2007*, J. Baumgartner and M. Sheeran, Eds. IEEE Computer Society, 2007, pp. 192–195.

[7] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodova, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel core i7 processor execution engine validation," in *Computer Aided Verification, CAV 2009: Proceedings*, ser. LNCS, A. Bouajjani and O. Maler, Eds. Springer-Verlag, 2009.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[9] J. Grundy, T. Melham, and J. O'Leary, "A reflective functional language for hardware design and theorem proving," *Journal of Functional Programming*, vol. 16, no. 2, pp. 157–196, Mar. 2006.

[10] *1800: IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language*. IEEE Computer Society, 2007.

[11] H. D. Foster and A. C. Krolnik, *Creating Assertion-Based IP*. Springer-Verlag, 2008.

[12] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *ACM/IEEE Design Automation Conference*. ACM Press, Jun. 1991, pp. 397–402.

[13] T. F. Melham and R. B. Jones, "Abstraction by symbolic indexing transformations," in *Formal Methods in Computer-Aided Design: 4th International Conference, FMCAD 2002*, ser. LNCS, M. D. Aagaard and J. W. O'Leary, Eds., vol. 2517. Springer-Verlag, 2002, pp. 1–18.

[14] S. Adams, M. Björk, T. Melham, and C.-J. Seger, "Automatic abstraction in symbolic trajectory evaluation," in *Formal Methods in Computer Aided Design: FMCAD 2007*, J. Baumgartner and M. Sheeran, Eds. IEEE Computer Society, 2007, pp. 127–135.

[15] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of Boolean constraints," in *ACM/IEEE Design Automation Conference*, Jul. 1998.

[16] L. Hellerman, "A catalog of three-variable Or-Invert and And-Invert logical circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, Jun 1963.

Data Mining Based Decomposition for Assume-Guarantee Reasoning

He Zhu*, Fei He*, William N. N. Hung†, Xiaoyu Song‡ and Ming Gu*
* Tsinghua National Laboratory for Information Science and Technology
Key Laboratory for Information System Security, Ministry of Education
School of Software, Tsinghua University, Beijing, China
Email: hefei@tsinghua.edu.cn
†Synopsys Inc., Mountain View, California, USA
‡Department of ECE, Portland State University, Oregon, USA

***Abstract*—Automated compositional reasoning using assume-guarantee rules plays a key role in large system verification. A vexing problem is to discover fine decomposition of system contributing to appropriate assumptions. We present an automatic decomposition approach in compositional reasoning verification. The method is based on data mining algorithms. An association rule algorithm is harnessed to discover the hidden rules among system variables. A hypergraph partitioning algorithm is proposed to incorporate these rules as weight constraints for system variable clustering. The experiments demonstrate that our strategy leads to order-of-magnitude speedup over previous.**

I. Introduction

Model checking is a popular formal verification technique. However, it suffers from the state explosion problem. Model checking of large scale systems such as microprocessors is known to be extremely difficult, and the problem will only get worse as the complexity scales exponentially with multi-core or distributed systems. To avoid the blow-up, people tend to utilize a divide-and-conquer strategy, i.e., decomposing large problems into multiple pieces and work on them separately. Assume-guarantee reasoning [1], [2] has been proposed as a useful technique to enhance model checking, especially when there are mutual dependencies between components. In this approach, individual component is verified separately according to the assumptions on its environment (i.e. other components), and then this assumption must be discharged by the rest of the system.

Assume-guarantee reasoning has been studied by many researchers for a long time [3], [4]. In this paper, we mainly use the following assume-guarantee rule [4]:

$$\frac{M_1||A \models \varphi \quad \text{(n1)} \qquad M_2 \sqsubseteq A \quad \text{(n2)}}{M_1||M_2 \models \varphi}$$

The rule above says that if M_1 combined with an assumption A satisfies the property φ (in n1 step), and A is further an abstraction of M_2 (in n2 step), then we conclude the system composed of M_1 and M_2 satisfies φ.

This work was supported in part by the Chinese National 973 Plan under grant No. 2004CB719400, the NSF of China under grants No. 60553002, 60635020, 60903030 and 90718039.

Assume-guarantee reasoning requires that correct assumptions be provided, which imposes additional hurdles for this method. [5] proposed a new method based on language learning to automatically learn assumptions from an alphabet over I/O (Input/Output) variables between components. However, this method may be insufficient when the construction of assumption needs to exhaust a large alphabet. In fact, the natural structure of components may be not applicable for system or modular verification. Hence, it is important to explore various decomposition boundaries that are independent of the original modular structure of the system.

Inspired by the work in [6], we present an effective automatic decomposition approach. Given a state transition system, the proposed method decomposes it into n sub-modules. We want to find a partition that converges to smaller assumption construction and early verification termination. In [6], the objective function is defined as partitioning the system to minimize the number of I/O variables between modules. In this paper, we modify the goal not only to reduce the I/O variables but also to enhance each module's cohesion.

If we define state variables' relation as distance which is a common measure concept in data mining, our partitioning goal is to minimize the intra-cluster distances within each module and to maximize the inter-cluster distances between modules. To quantify the distances we propose to use association rule mining to reveal the hidden variable implications and provide qualified information for decomposition. In this way, we find a partition which put state variables frequently communicate to each other together.

To evaluate our method, we incorporate the NuSMV model checker [7] with a synchronous model checker SYMODA [8]. Experimental results show some encouraging improvements of our approach over previous [6].

II. Preliminaries

In this section, we define the concept of decomposition of a state transition system and composition which are used through our paper. Our formalisms use notations similar to [6].

Given any set of state variables X, for each $x \in X$, x is a typed variable defined over a finite domain of values D_x. An assignment $s : X \to V$ maps each x in X to one certain value

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

v in D_x. We write boolean formula $\varphi(s)$ for a property over X and we say $\varphi(s)$ is true if the assignment $s(X)$ satisfies φ. We use $Var(\varphi)$ to denote the set of state variables appearing in φ in our context.

Formally, a state transition system S is a tuple $\langle X, Init_X, T_X\rangle$, where

1) X is a set of state variables of the system.
2) $Init_X = \bigwedge_{x\in X} Init_x(X)$ is an initial predicate over X where $Init_x(X)$ is an initial predicate for variable x.
3) $T_X = \bigwedge_{x\in X} T_x(X, X')$ is a transition predicate over X and X', where X' denotes the next statues of X and $T_x(X, X')$ is the transition predicate for variable x.

A state transition system S may comprise several sub-modules. A sub-module M_i is a tuple $\langle X_i, I_{X_i}, O_{X_i}, Init_{X_i}, T_{X_i}\rangle$, where

- $X_i \subseteq X$ is a set of state variables controlled by M_i.
- I_{X_i} is a set of input variables that are controlled by some other modules and are readable by M_i. Note I_{X_i} is disjoint from X_i.
- $O_{X_i} \subseteq X_i$ is a set of output variables that are controlled by M_i and are accessible by some other modules.
- $Init_{X_i} = \bigwedge_{x\in X_i} Init_x(X)$ is an initial predicate over $X_i \cup I_{X_i}$.
- $T_{X_i} = \bigwedge_{x\in X_i} T_x(X, X')$ is a transition predicate over $X_i \cup X_i' \cup I_{X_i}$.

We use IO_{X_i} to denote the input and output variables of M_i, i.e. $IO_{X_i} = I_{X_i} \cup O_{X_i}$.

The semantic of a state transition system is the set of runs it exhibits. A run of S is a sequence $s_0, s_1, \ldots$ of states where each s_i represents a variable assignment mapping each value in X to its domain, such that $Init_X(s_0)$ holds and for every $j \geq 0$, $T_X(s_j, s_{j+1})$ holds.

Given a state transition system $S(X, Init_X, T_X)$ and an integer n, decomposition problem is to decompose S into n sub-modules $M_i(X_i, I_{X_i}, O_{X_i}, Init_{X_i}, T_{X_i})$, $1 \leq i \leq n$, where $X = \bigcup_{1\leq i\leq n} X_i$ and $\bigwedge_{i,j=1,\ldots,n}^{i\neq j} X_i \cap X_j = \emptyset$.

Given a property φ over $\cup_{1\leq i\leq n} IO_{X_i}$, let $S \models \varphi$ denote φ holds in S, i.e., for each run $s_0, s_1, \ldots$ of S, $\varphi(s_0), \varphi(s_1), \ldots$ holds. According to [6], there is

$$(S \models \varphi) \Leftrightarrow (M_1 || \ldots || M_n \models \varphi).$$

III. Problem Formulation

In this section, we formulate the system decomposition problem into a hypergraph partitioning problem.

A. Hypergraph Partitioning

A hypergraph is a special graph, which can be defined as $G(V, E)$, where

- V is a set of vertices.
- E is a set of hyperedges that connect arbitrary number of vertices. For a hyperedge $e \in E$, let $vertex(e)$ denote the set of vertexes connected by e.

A weighted hypergraph is one that assigns each hyperedge a numerical value. More formally, a weighted hypergraph is a triple $G(V, E, W)$, where V, E are defined identically as in normal hypergraph, and $W : E \to \mathbb{R}$ defines a weight value for each hyperedge.

The K-way hypergraph partitioning problem $P(G, K, \lambda)$ is to partition the original hypergraph G into K parts by clustering the vertexes of G into K disjoint subsets $V_1, \ldots, V_K$, such that $V = \bigcup_{1\leq i\leq K} V_i$. After partitioning, a hyperedge may span different parts. The concept of connectivity λ helps to distinguish whether a hyperedge crosses some parts or lies inside one single part. Connectivity λ_j of a hyperedge $e_j \in E$ is 1 if the number of different parts e_j crosses is greater than 1; otherwise λ_j is 0. A hyperedge e_j is called as a hyperedge-cut if $\lambda_j = 1$.

The K-way hypergraph partitioning algorithm tries to find a partition to minimize $C_p = \sum_{e_j\in E} w_j \cdot \lambda_j$, where w_j is the weight of hyperedge e_j. It also imposes a constraint that the cardinality of each set V_i is bounded by $|V|/(cK) \leq |V_i| \leq |V|(c/K)$ where $|V|$ is the number of vertexes contained in V. The imbalance tolerance c is a parameter outside the partitioning algorithm. A large value of c causes imbalance clusters while a small value of c makes each V_i roughly the same size.

Many researchers have studied hypergraph partitioning intensively and there already exists fast algorithms and tools. Among them we choose hMETIS [9].

B. Decomposition as Hypergraph Partitioning

Given a state transition system $S(X, Init_X, T_X)$, we say there exists variable dependency between x and x', if x' appears in $T_x(X, X')$. Formally, let Y denote the set of variables appearing in $T_x(X, X')$, the variable dependencies of x is the power set of Y.

Given a hypergraph $G(V_X, E)$, we call a vertex v_y is an adjacency of another vertex v_x if it is connected to v_x by some hyperedge in E. The vertex adjacencies of v_x is a set in which each element is a set of vertexes connected by a hyperedge involving v_x i.e., $\bigcup_{e\in E}\{vertex(e)|v_x \in vertex(e)\}$.

Based on above points, if the state variables in X are linked to the vertices in V_X, the power set of variable dependencies of a variable x can be modeled as hyperedges involving v_x. Furthermore, the power sets of variable dependencies certainly have different occurrence in system's state transitions. Intuitively some sets of variables occur more times than the other sets in T_X. If one applies certain precise measure to assign each subset of variable dependencies a numerical value, the hypergraph $G(V_X, E)$ can be upgraded to a weighted hypergraph $G(V_X, E, W)$.

IV. Data Mining Based Decomposition

The flow of our method is shown in Fig. 1. In essence, a state transition system is modeled by a weighted hypergraph then the partition of hypergraph model uniquely determines the system decomposition.

The challenge in our method is how to measure weight values for each hyperedge in the hypergraph model. In this paper, we propose a novel method by applying data mining algorithm to generate these weight values.

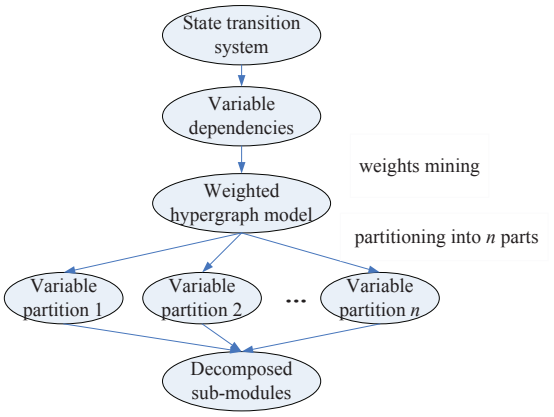


Fig. 1. Our Decompsition Method

A. Association Rule Mining

Association rule mining [10] aims to discover the hidden patterns and correlations from a large dataset. Given an itemset $I = \{I_1, I_2, \ldots, I_n\}$ and a set of transactions T where $t \in T$ is a subset of I. Let X, Y denote two disjoint subsets of I, i.e. $X, Y \subseteq I$, $X \cap Y = \emptyset$. An association rule is an implication in the form of $X \Rightarrow Y$. We define a function $f : 2^I \rightarrow \mathbb{N}$ mapping each subset of I to a natural number which represents the number of transactions (in T) containing this subset.

An association rule is defined by two important notions, the support and confidence of the rule. The support of a rule is defined as $sup(X \cup Y) = f(X \cup Y)/|T|$, i.e. the percentage of transactions that contain $X \cup Y$, where $|T|$ is the total number of transactions contained in T. While the confidence of a rule is defined as $conf(X \cup Y) = f(X \cup Y)/f(X)$, i.e. the ratio of number of transactions that contain $X \cup Y$ to the number of transactions that contain X.

Given a set of transactions T, association rule mining first finds frequent itemsets by generating all combinations of items in I with their supports above user supplied $minsup$ threshold. For a frequent itemset fi, all the association rules with the type $X \Rightarrow fi - X$ $(X \subset fi)$ are outputted if their confidences are above user supplied $minconf$ threshold.

There are many association rule mining algorithms, among which Apriori [10] is a fast and popular algorithm. We use the Apriori algorithm implemented in [11].

B. Weights Mining for Hypergraph Model

Given a state transition system $S(X, Init_X, T_X)$, we convert it into a weighted hypergraph $G(V_X, E, W)$. There are two steps in mining hyperedge weights from S.

1) First step: collect variable dependencies as transactions: In this step, we generate the transactions as the input of association rule mining algorithm by deriving variable dependencies from the system's state transitions.

For each $x \in X$, there is a transaction t_x containing variable x and a variable y belongs to t_x if $y \in Var(Init_x) \cup Var(T_x)$. Intuitively, t_x represents the corresponding variable x and all variables that are read by x. The transactions for S is finally the set of $V_T = \{t_x | x \in X\}$. We apply Apriori algorithm to transactions V_T to generate frequent itemsets and association rules upon X. Suppose the generated frequent itemset is $E_{f_i} = \{f_i | f_i \subseteq t_x \wedge t_x \in V_T \wedge sup(f_i) > minsup\}$.

2) Second step: combine association rules with weighted hypergraph model: In this step, we provide a precise measure to assign each hyperedge a numerical value which is explained in section III.

Each variable x in X corresponds to a vertex v_x in V_X such that $V_X = \{v_x | x \in X\}$. For each frequent itemset $f_i \in E_{f_i}$, we make a hyperedge e_{f_i} and a vertex v_x is connected by e_{f_i} if its corresponding variable $x \in f_i$. The weight of the hyperedge e_{f_i} is the average of confidences of all the association rules derived from the frequent itemset f_i, i.e., $W(e_{f_i}) = \forall_{X \subset f_i} \overline{\{conf(X, f_i - X) | conf(X, f_i - X) > minconf\}}$.

We apply hypergraph partitioning algorithm to the generated weighted hypergraph model. After partitioning, we classify variables $X_1, \ldots, X_n$ according to the vertex partitioning result $V_1, \ldots, V_n$. Then we use the results in section II to build each sub-module $M_i(X_i, I_{X_i}, O_{X_i}, Init_{X_i}, T_{X_i})$, where $1 \leq i \leq n$.

Note there are two intrinsic differences between our method and the counterpart of [6] in modeling a hyperedge. First, given a variable x, let x and all the variables that read x compose a set Y; let x and all the variables that are read by x compose Z. The method in [6] models the whole set Y as a hyperedge; while our method models the subsets of Z as corresponding hyperedges. Second, our method incorporates numerical value into hyperedge by using association rules.

Association rule mining is applied to help us find groups of variables with dense dependencies. The generated rules with high confidence indicate the variables referenced by the rules have close dependency between them. While the rules with low confidence indicate the variables referenced by the rules have sparse correlation. Thus in our weighted hypergraph model, heavy weighted hyperedges are more likely to be partitioned in the same cluster while light weighted hyperedges are more likely to be the hyperedge-cuts.

According to the assume-guarantee rule, we need to verify $M_1 || A \models \varphi$ (n1) and $M_2 \sqsubseteq A$ (n2). Suppose we partition the system model into two modules and assign M_1 to be the module which manages almost or all the variables in $Var(\varphi)$. As M_1 has compact state space related to φ, an appropriate assumption A which satisfies $M_1 || A \models \varphi$ (n1) can then be constructed with fewer interventions from M_2. In this case, A avoids to be strengthened too much times to prevent M_1 from getting to the error state and the state space to be visited is reduced. On the other hand, as M_1 and M_2 have relatively few communications, $M_2 \sqsubseteq A$ (n2) can also be verified readily because we have very loose bound on assumption A and A is weak enough in this case.

V. Experiment Results

We have implemented our method and integrated it with the symbolic model checkers NuSMV [7] and SYMODA [8]. We use NuSMV as a front-end to parse NuSMV model file and

TABLE I
EXPERIMENTAL RESULTS

Example	var	Weighted Hypergraph						Unweighted Hypergraph						General
		minsup	c	IO	Mem	Candi	Total	minsup	c	IO	Mem	Candi	Total	Total
s1a	23	0.05	1.0	2	0.03	0.17	0.32	0.05	1.0	2	0.04	0.15	0.31	15.77
s1b	25	0.05	1.0	6	0.14	0.07	0.49	0.05	1.0	6	0.20	0.12	0.60	16.03
msi3	61	0.05	1.8	17	0.90	0.22	2.81	0.05	1.8	19	1.20	0.66	3.53	10.23
msi5	97	0.05	1.8	24	2.31	0.40	5.86	0.05	1.8	32	3.62	1.41	8.81	27.17
msi6	121	0.05	1.8	27	3.42	0.55	9.69	0.05	1.8	33	5.16	1.36	12.11	43.80
syncarb10	74	0.01	1.0	32	26.85	23.18	76.13	0.01	1.0	33	68.94	46.18	129.2	TO
peterson	9	0.01	1.0	7	0.18	0.31	0.65	0.01	1.0	7	19.89	70.34	113.8	27.67
guidance	76	0.05	1.0	37	11.19	4.11	19.93	0.05	1.0	13	1.03	2.00	4.11	18.75

generate sub-modules by our weighted hypergraph model. We use Symoda, a synchronous modular analyzer, to verify the above generated decomposed modules. All experiments were performed on a 1GB memory 3 GHz Intel machine running Ubuntu Linux System.

All benchamrks are obtained form the public places [6], [12], [13]. Table I compares the decomposition results by verification time for our weighted hypergraph model aided, unweighted hypergraph model [6] aided verification and general model checking without decomposition. All examples are partitioned into two components.

In Table I, $minsup$ is a parameter for Apriori. The other threshold $minconf$ is always assigned to its default value (0.8). c is a parameter for hMETIS. We try five different values (i.e. 1.0, 1.2, ..., 1.8) and pick one that is suitable for each benchmark. var represents the number of variables. IO gives the input/output variables between sub-modules. Note we count up the number of typed variables instead of boolean variables. Except $guidance$, all benchmarks have more than (or about) one hundred boolean variables. Mem and $Candi$ are the total membership and candidate query time respectively. $Total$ gives the overall verification time. All times are in seconds. TO means verification doesn't terminate in 1200s.

It is obvious that assume-guarantee reasoning with appropriate decomposition performs much better than the general modular method without decomposition as the decomposition either reduced verification time or converted infeasible problem into feasible one. In most cases, our solution that incorporates association rule mining and weighted hypergraph model obtains significantly better result than the approach in [6] which used unweighted hypergraph partitioning method solely. In $peterson$ and $syncarb10$ examples, our solution achieves dramatic improvements.

However our method obtained negative result as shown in $guidance$ example. It is probably because the transition relations between the variables in original $guidance$ model are so sparse, i.e., a large portion of variables in this example only interact with no more than two variables directly. In this case, Apriori algorithm can't mine meaningful rules but leads to inappropriate decomposition.

VI. CONCLUSION

In this paper, we presented an automatic method to decompose system model into simpler sub-modules. The use of association rule mining improves the partitioned modules' inner cohesion and the confidence of association rules restricts to put related variables together. As a result, inter-connections between modules are reduced and each module is compact. The experiments demonstrate our method not only leads to good decomposition but also improves verification scalability in the context of assume-guarantee reasoning.

We used non-circular assume-guarantee rule in this paper. In future, we plan to extend our framework to make it scalable to circular assume-guarantee rules [14] too. Furthermore, data mining remains an active subject; we also intend to apply some other classification algorithms to find even better system decompositions contributing to simple assumptions.

REFERENCES

[1] A. Pnueli, "In transition from global to modular temporal reasoning about programs," *Logics and models of concurrent systems*, 1985.

[2] C. B. Jones., "Specification and design of (parallel) programs," in *Proceedings of the IFIP 9th World Congress*, 1983, p. 321332.

[3] T. Henzinger, S. Qadeer, and S. Rajamani, "You assume, we guarantee: Methodology and case studies," in *Proc. Computer Aided Verification (CAV)*, 1998, pp. 521–525.

[4] K. S. Namjoshi and R. J. Trefler, "On the completeness of compositional reasoning," in *Proc. Computer Aided Verification (CAV)*, 2000, pp. 139–153.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[6] W. Nam and R. Alur, "Learning-based symbolic assume-guarantee reasoning with automatic decomposition," in *Automated Technology for Verification and Analysis (ATVA)*, 2006, pp. 170–185.

[7] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking," in *Proc. Computer Aided Verification (CAV)*, 2002, pp. 359–364.

[8] N. Sinha and E. Clarke, "Sat-based composition verification using lazy learning," in *Proc. Computer Aided Verification (CAV)*, 2007, pp. 3–5.

[9] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain," *IEEE Trans. VLSI Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[10] R. Agrawal, T. Imielienski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," in *Proc. Conf. Management of Data*, 1993, pp. 207–216.

[11] C. Borgelt, "Efficient implementations of apriori and eclat," in *Proc. ICDM workshop on frequent itemset mining implementations*, 2003.

[12] NuSMV examples: the collection. [Online]. Available: http://nusmv.irst.itc.it/examples/examples.html

[13] VIS verification benchmarks. [Online]. Available: ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.0.tar.gz

[14] H. Barringer, D. Giannakopoulou, and C. S. Pasareanu, "Proof rules for automated compositional verification," in *2nd Workshop on Specification and Verification of Component-Based Systems (ESEC/FSE)*, 2003.

Scalable Conditional Equivalence Checking: An Automated Invariant-Generation Based Approach

Jason Baumgartner[1] Hari Mony[1] Michael Case[1] Jun Sawada[2] Karen Yorav[3]
[1]IBM Systems & Technology Group [2]IBM Austin Research Laboratory [3]IBM Haifa Research Laboratory

***Abstract*—Sequential equivalence checking (SEC) technologies, capable of demonstrating the behavioral equivalence of two designs, have grown dramatically in capacity over the past decades. The ability to efficiently identify and leverage *internal equivalence points* to reduce the domain of the overall SEC problem is central to SEC scalability. However, *conditionally equivalent* designs – within which internal equivalence may not exist under sequential *observability don't care* conditions – are notoriously difficult for automated SEC tools. This paper constitutes one of the first attempts to advance the scalability of SEC for conditionally equivalent designs through automated invariant generation, which enables an inductive solution to an otherwise highly-noninductive problem. Through careful software engineering and various heuristics, this technique has been demonstrated capable of yielding orders of magnitude speedup on difficult industrial conditional SEC problems, in cases constituting *the only method* that we have found to achieve an automated solution.**

I. Introduction

Equivalence checking refers to the process of demonstrating the behavioral input-to-output equivalence of two designs. Numerous equivalence checking paradigms exist in practice. Combinational equivalence checking (CEC) is a framework where the state elements of two designs have 1:1 correlation. Instead of directly checking input-to-output equivalence, CEC frameworks *assume* that correlated state elements are equivalent, and demonstrate that outputs – as well as next-state functions of the correlated state elements – are equivalent. CEC frameworks thus avoid computationally-expensive sequential reasoning by decomposing the overall equivalence check into a set of combinational proof obligations [1]. Due to ease of use and scalability, CEC is pervasively used throughout the semiconductor industry – for example, to validate that logic synthesis preserves the behavior of the original design.

Sequential equivalence checking (SEC) is a generalization of CEC wherein the designs being equivalence checked may not have a 1:1 state element correlation [1]. The benefit of SEC over CEC is obvious: if a *sequential* transformation is performed across the designs being equivalence checked, CEC is no longer directly applicable – at least, not without substantial manual or restrictive methodological guidance. Such sequential transformations – e.g., retiming, state re-encoding, unreachable-state based optimizations, etc. – are commonly used in the design of high-performance circuits. Due to its generality, SEC generally requires analysis of the sequential behavior of the designs being equivalence checked – and thus comes with substantially greater computational expense. To be precise, CEC solves a set of subproblems in class NP, whereas SEC solves a more monolithic problem in class PSPACE [2].

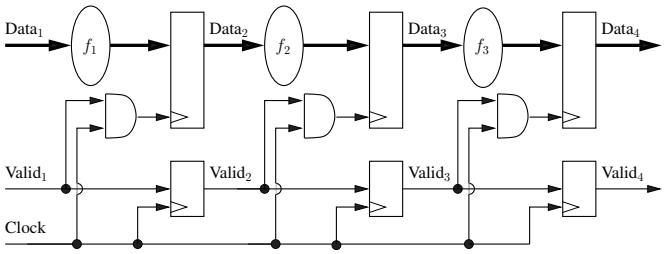


Fig. 1: Clock-gated pipeline

The ability to leverage internal equivalence points is often critical to the scalability of SEC. As with CEC, instead of merely demonstrating input-to-output equivalence, a set of internal equivalences may be demonstrated in conjunction. This overall set of properties is often substantially easier to solve than direct input-to-output equivalence: invariants that stipulate internal equivalences often enhance inductivity by strengthening the collective induction hypothesis [3], [4]. However, unlike CEC, it is generally not the case that every state element constitutes an internal equivalence point; the two designs may not have 1:1 correlation of state elements, nor behavioral equivalence among those which appear correlated (e.g., having identical signal names in their HDL definition).

Speculative reduction is often used to speed the overall SEC process. This technique consists of merging fanout references to suspected-equivalent gates even before they are proven equivalent, thereby simplifying proof goals expressed over this fanout logic. To ensure soundness, the validity of the speculatively-merged gates is checked as a set of additional proof obligations, and the SEC process attempts to solve all proof goals in conjunction. It has been demonstrated that the well-architected use of speculative reduction enables up to five orders of magnitude speedup on difficult industrial SEC problems, often making the difference on whether or not a conclusive result may be obtained [5], [6]. While SEC remains a problem in PSPACE as internal equivalence points may span arbitrary sequentially-redesigned subcircuits, speculative reduction often enables lighter-weight algorithms such as structural simplification and induction to solve problems which otherwise would require heavier-weight proof techniques.

In this paper, we address a generalization of SEC: *conditional sequential equivalence checking* (CSEC). Unlike the above-mentioned SEC paradigm, wherein equivalence is checked at all points in time and across all execution sequences, CSEC allows designs to depart from equivalent behavior under specific time-frames. For example, consider

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

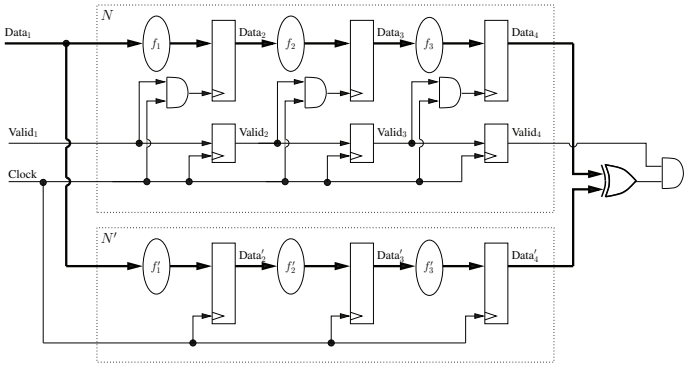


Fig. 2: Clock-gated CSEC testbench

the design of Figure 1 which pipelines its data computation across three clock periods. When a given pipeline stage is not required to produce a valid result, its clock may be disabled to reduce power consumption: a low-power technique called *clock gating* [7]. It is often desirable to validate that a version of the design with clock gating disabled (or not integrated) produces equivalent results to a version with clock gating enabled, to ensure that clock gating does not inadvertently alter design behavior. This is depicted in Figure 2, where N is equivalent to Figure 1 and N' is a simplified version without clock gating. Note that this is a CSEC problem: when no valid instruction is being processed ($\text{Valid}_i \equiv 0$), N will hold the results of its last computation whereas N' will needlessly process whatever invalid data is present at its inputs. The equivalence check over Data_4 vs. Data'_4 is thus restricted to conditions where valid output is required from N.

Power gating is a related CSEC problem domain: when a design component is idle, its voltage supply may be disabled by a controller. Power is restored only when that controller detects an imminent processing need [7]. Power-disabling is often modeled in verification by randomizing or "tristating" state element contents when their voltage is disabled [8].

CSEC problems are notoriously difficult to solve, since they lack the internal equivalences which are key to the scalability of traditional SEC approaches. While internal equivalences no longer unconditionally hold in CSEC problems, it is often the case for correct designs that internal equivalences hold *conditionally*. For example, in Figure 2, when $\text{Valid}_i \equiv 1$, the state elements of Data_i vs. Data'_i are pairwise equivalent. Conversely, the only time-frames at which corresponding state elements are inequivalent is an *observability don't care* (ODC) condition, which may never propagate to the outputs of the design at relevant time-frames. Given an adequate set of conditional equivalence invariants, it is often the case that the CSEC problem becomes k-inductive whereas otherwise it would not be. This observation is the motivation of this paper: to automate the derivation of an adequate set of invariants to in turn enable an efficient automated CSEC solution. We furthermore assume a synthesis-unaware paradigm for maximal applicability and generality, e.g., in case such optimizations are performed manually. We thus do not rely upon synthesis-history hints (e.g., [9]) for applicability of our techniques, though such hints may readily be incorporated into our framework to further enhance its scalability.

II. Preliminaries

The domain of the equivalence check is assumed to be over two hardware designs represented as *netlists*.

Definition 1: A *netlist* is a tuple $\langle\langle V, E\rangle, G, I, O, M\rangle$ comprising a finite directed graph with vertices V and edges $E \subseteq V \times V$. Function $G : V \mapsto types$ represents a mapping from vertices to $gate\ types$, including primary inputs $I \subseteq V$, state elements which each have an associated *initial-* and *next-state function*, and combinational gates with various functions.[1] Set $O \subseteq V$ represents the primary outputs, and set $M \subseteq V \setminus \{I, O\}$ represents the visible internal gates.

Netlists are often analyzed using *Boolean modeling*, wherein every gate evaluates to a Boolean value over time. Alternatively, *ternary modeling* may be used wherein gates take values from the set $\{0, 1, X\}$, where X refers to an "unknown" value. A gate that evaluates to a ternary X value is referred to as *tristated*.

Definition 2: An *equivalence checking testbench* comprises a netlist N'' which is the composition of two netlists N and N' under bijective mapping $I \mapsto I'$, along with possibly additional testbench logic; bijective mappings $O \mapsto O'$ and $M \mapsto M'$; and an *equivalence condition mapping* $C'' : O \mapsto V''$. The composition of N and N' is such that correlated elements of I and I' become merged as a single primary input. Mapping C'' defines the equivalence checking objectives for the testbench, i.e., a property $\big(C''(o) \rightarrow (o \equiv o')\big)$ to check of each correlated output pair $\langle o, o'\rangle$ of $\langle O, O'\rangle$.

Testbench logic refers to gates introduced to a netlist solely for verification purposes. E.g., any necessary input constraints may be synthesized and composed with the primary inputs of the netlist, or be enumerated with designated language constructs [10]. Similarly, logic may be introduced during synthesis of properties to be checked of the netlist, e.g., to specify equivalence conditions C''. For SEC, the range of C'' is simply $\{\top\}$: correlated outputs are checked for equivalence at all time-frames. For CSEC, C'' indicates at which points in time – and along which execution streams – output equivalence is to be checked (e.g., Valid_4 in Figure 2).

Note that C'' along with mappings $I \mapsto I'$ and $O \mapsto O'$ are *required* to establish the semantics of an equivalence checking testbench. Practically, mapping $M \mapsto M'$ is often readily available for the problem domains of interest since the designs being equivalence checked are often identical modulo the addition or enabling of extra circuitry, which is the optimization that induced the conditional equivalence. Examples include power gating and clock gating as discussed above, or general *sequential ODC*-based optimizations, wherein portions of the

[1] In this definition, we assume that gate behavior is independent of input permutation: for example, state elements with distinct *clock* and *data* inputs have been synthesized into simple one-input delay gates with adjacent combinational logic to preserve clocking semantics. This definition may be straight-forwardly extended to more general netlist types if desired.

designs may produce different results under conditions where those differences do not propagate to a failed equivalence check. We discuss the relevance of this mapping in the following section.

A. Invariants

An *invariant* is a property of a netlist which holds in all reachable states. An invariant may be represented through added circuitry as a gate which evaluates to $\top$ in every reachable state. While a functionally redundant characterization of netlist behavior, an invariant may be used to reduce the degree of overapproximation of certain proof techniques, and thus enable a more efficient solution. For example, k-induction is a proof framework that attempts to demonstrate that no state (reachable or not) which cannot violate a property within k time-frames may do so in greater than k time-frames [11]. The overapproximation inherent in induction is that if this particular check fails, one generally cannot determine whether the failing "inductive state" is actually reachable or not. Similarly, interpolation is a framework which overapproximates the reachability analysis of a netlist [12], risking the appearance that some property-violating unreachable states are actually reachable. Invariants may be used to enhance such frameworks because they constrain the overapproximation toward exact netlist behavior [4], [13].

Numerous techniques have been proposed for automated invariant generation. Examples include those which derive constant and equivalent gate pairs [3], [4]; those which derive implications among gates [4], [14]; those which attempt to enumerate invariants expressible using k-literal clauses [15], [16]; and heavier-weight techniques which perform approximate reachability analysis [13]. Some invariant learning approaches are *eager* in attempting to derive arbitrary invariants of a certain characterization [4], [15], [16], [14], while others are *lazy* in attempting to establish only invariants which will preclude spurious property failures in an overapproximating verification framework [17], [18]. A general tradeoff which must be considered is how much effort to devote to invariant generation vs. to spend directly attempting to solve the problem under consideration. Of even greater importance is the tailoring of the invariant generation process to the problem domain being addressed.

In this paper, we propose an eager framework for invariant generation tuned for enabling complex CSEC problems to be efficiently solved using overapproximating proof techniques such as induction and interpolation. In particular, given an equivalence checking netlist N'', our goal is to derive a set of conditional equivalence invariants of the form $\big(g_i \rightarrow (m_i \equiv m_i')\big)$ for some gate g_i and *correlated-gate pair*[2] $(m_i, m_i') \in M''$ to enable an efficient proof of the CSEC objectives. We refer to the set of gates g_i for which $(m_i \equiv m_i')$ as the *equivalence conditions* of m_i and m_i', denoted $E(m_i)$ or equivalently $E(m_i')$. Collectively, the set of derived invariants are $\big\{\forall (m_i, m_i') \in M''.\forall g_i \in E(m_i).\big(g_i \rightarrow (m_i \equiv m_i')\big)\big\}$.

[2]We hereafter abuse notation, referring to $M \mapsto M'$ as the relation M''.

Generally, one need not limit conditional equivalence invariants to this syntax: one may attempt to derive invariants over arbitrary gate triples, for example. However, such an approach is unattractive for industrially-relevant netlists because: **(1)** there are a *cubic* number of such invariants possible, and implication analysis – which entails only a *quadratic* number of candidate invariants – is known to face scalability challenges hence occasionally must be performed lossily [14]. The proposed syntax reduces the number of candidate invariants to quadratic. **(2)** As per the clock-gating and power-gating examples of Section I, this syntax is adequate to capture a suitably-expressive set of invariants to enable the efficient solution of otherwise highly-intractable CSEC problems. **(3)** We have developed numerous heuristics to expedite the eager derivation of candidate invariants of this form. In contrast to eager generation, we have found lazy generation of such invariants – e.g., based upon conditions witnessed from induction counterexamples [17] – to be of comparable or even greater computational expense due to the large number of candidate invariants of this form.

Additionally, note that the intrinsic purpose of our invariant generation framework is to cope with ODC-based optimizations across the netlists being equivalence checked. Despite a large amount of work in practical ODC derivation (e.g., [19], [20]), such approaches grow intractable in computational complexity as the logic windows against which ODC conditions are computed grow in size. The nature of the CSEC problem generally relies upon ODCs which span the majority of the netlist, thus rendering direct ODC computation to be an impractical approach to enabling the solution of CSEC problems, and motivating our invariant generation approach.

III. Algorithms

In this section we discuss our algorithms for computing conditional equivalence invariants. Our basic flow is depicted in Figure 3, and is similar to those used for traditional SEC, e.g. [3], [5]. In particular, we first compute a set of candidate invariants in Step 1, some of which may be invalid. In Step 2 we may use underapproximate analysis techniques such as random simulation and bounded model checking [21] to efficiently eliminate invalid candidates. We then iterate a proof phase attempting to validate the remaining invariants in Step 3. If any candidate invariants cannot be proven, due to being incorrect or due to being computationally intractable, those invariants are eliminated and another proof is attempted over those which remain in Step 4. The reason for iterating this check until all candidates are proven together is that it is often desirable to cross-leverage each invariant to tighten any overapproximate analysis used to prove the other invariants, e.g., to strengthen a collective induction hypothesis. Any invalid candidates may jeopardize the soundness of other proof results in such a framework.

There are two competing objectives in the choice of candidate invariants in Step 1. First: by considering each gate g in the equivalence checking netlist as a possible equivalence condition for each element of M'', a quadratic number of

1. Derive a set of candidate conditional equivalence invariants $E(m_i)$ for each $(m_i, m_i') \in M''$
2. Optionally utilize underapproximate analysis to falsify invalid invariants, pruning E
3. Attempt to prove that each of the candidate invariants is accurate
4. If any invariants cannot be proven, prune them from E; goto Step 3
5. All invariants have been proven accurate; return the resulting E

Fig. 3: Conditional equivalence invariant generation

invariants may be considered in this framework. For large netlists, direct consideration of such a large number of candidate invariants may require exorbitant resources. We address this concern in Section III-B. Second, the goal of this invariant generation process is to yield an adequate invariant set E to enable a subsequent efficient overapproximate proof technique – ideally induction – to solve the resulting CSEC problem. Approaches to subset the candidate invariants jeopardize the derivation of an adequate set of invariants for this purpose.

Note that merely considering every gate in the netlist as a possible equivalence condition is theoretically lossy in itself, in that a condition under which a particular conditional equivalence holds need not exist in the form of a gate in the netlist. We have found this a minor concern in practice. In applications such as clock gating and power gating, the necessary conditions often appear in the design in the form of gates that are added as an essential part of the optimization itself. In many other cases, the necessary conditions naturally appear in the synthesized testbench logic, since this logic specifies when output equivalences are to be checked. Finally, when ternary modeling is used, to model power gating, an undriven bus, an undefined *case* condition, or other *don't care* conditions [22], an often-adequate condition is to check equivalence whenever a correlated-gate pair is *not tristated*.

A. Computing Candidate Invariants

There are numerous problem domains where the number of candidate invariants of relevance may be contained to constant on a per correlated-gate basis. In particular, consider the use of ternary modeling wherein the X-value reflects the only conditions under which portions of the netlists being equivalence checked may differ. A linear number of candidate invariants may be derived using the procedure of Figure 4. Examples of such cases include power-gating verification (refer to Section IV-B), as well as cases where ternary X may be used to model don't care conditions or floating buses [22]. Another example is *uninitialized* sequential equivalence checking (e.g. *alignability analysis* [23]), which may be modeled by initializing all state elements with a ternary X, then driving a *weakly synchronizing input sequence* that brings the equivalence checking netlist into a state containing fewer X'es from which the outputs are pairwise equivalent and Boolean-valued [24].

In other problem domains, a potentially quadratic number of candidate invariants may need to be considered. In these cases, the practicality of our invariant generation framework is critically dependent upon the use of efficient algorithms and careful software engineering. Our solution has the following characteristics.

1. Build a ternary-modeled equivalence checking testbench
2. For every $(m_i, m_i') \in M''$, add candidate invariants into E of the form $\{(\neg tristated(m_i) \rightarrow (m_i \equiv m_i')), (\neg tristated(m_i') \rightarrow (m_i \equiv m_i'))\}$
3. Goto Step 2 of the algorithm of Figure 3 to obtain the final accurate E

Fig. 4: Ternary conditional equivalence candidate generation

(1) As illustrated in Figure 3, our basic flow is to first associate a set of candidate equivalence conditions with each gate pair in M'', then to use underapproximate analysis techniques to eliminate many of the invalid candidates, and finally to attempt to prove the remaining candidate invariants correct. We represent the candidate invariants using a *trie* data structure, which stores sets of candidate equivalence conditions for each $(m_i, m_i') \in M''$. The benefits of using a trie are that each unique equivalence condition set only requires a single data representation, and more importantly that common subsets of candidates across different correlation pairs may share in their data representation.

(2) When one counterexample trace is obtained (typically using incremental SAT) which invalidates one candidate invariant, we use a highly-tuned random simulation process to efficiently rule out large sets of invalid invariants, similar to what is done for traditional SAT-sweeping [25]. In particular, we use a *bit-parallel simulator* which models the behavior of the netlist across a variety of randomized input patterns, though seeded to adhere to the behavior witnessed in the original counterexample. By using 1024 parallel simulation patterns, and by extending the length of this random simulation by several time-frames with respect to the original counterexample length, we commonly witness ratios of 100:1 to 100000:1 in terms of the number of invalid candidates ruled out by re-simulation of SAT traces vs. the number of satisfiable SAT calls themselves. We have tuned the integration of the bit-parallel simulator with the trie data structure so that incorrect candidates are pruned as efficiently as possible.

Overall, while in the worst case a quadratic number of candidates may need to be considered, through careful software engineering we have reduced the memory overhead and runtime of our conditional equivalence invariant generation framework to approach linearity as closely as possible.

B. Subsetting Candidate Invariants

An additional motivation for efficiency is to attempt to minimize the number of possibly valid candidate invariants, while nonetheless ensuring that the resulting set preserves overall CSEC inductiveness. We have found the following heuristics useful for this purpose, which are selectively applicable on a per-problem-domain basis. Regarding which of these heuristics to select for a particular CSEC problem: note that information about the nature of the CSEC testbench may often yield insight into which of these heuristics (or the ternary conditional equivalence approach of Figure 4) may be most applicable. Additionally, note that in a fully-automated flow, these various algorithmic flavors may be run in parallel to the fully-quadratic algorithm, enabling convergence to a conclusive solution with a minimum of elapsed walltime. Automated sharing of invari-

1. for (i=0; ¬*terminate*; i++)
2. T_i = set of gates which may toggle at time i along any trace
3. M_i = gate pairs from M'' which may mismatch (or toggle from a mismatch to a match) at time i along any trace
4. Enumerate T_i as candidates for M_i in E
5. Validate E at time i
6. Goto Step 2 of the algo of Figure 3 to obtain the final accurate E

Fig. 5: Toggle-based equivalence candidate generation

ants across these various algorithms is also possible, which may incrementally approach the exhaustiveness of the fully-quadratic algorithm while benefitting from the speed of the subsetting techniques.

(1) One often may meaningfully subset the set of candidates on a per correlated-gate basis by analyzing the sequential behavior of the netlist. For example, considering the pipelined example of Figure 1, intuitively the logic comprising pipeline stage i would not constitute useful conditions for pipeline stages $j \neq i$. For general cyclic sequential netlists, the question of *which* gates may be effective conditions for *which* correlated-gate pairs becomes complex due to the inability to meaningfully structurally associate "pipeline stages." Our solution to this problem is to analyze *toggle* and *mismatch* activities of equivalence condition candidates and correlated-gate pairs, as depicted in Figure 5. Steps 2, 3, and 5 may generally be performed using a combination of incremental SAT and random simulation.

For certain types of designs, e.g. pipelines such as Figure 1, merely checking which gates may first toggle to a particular value at time i (set T_i), and associating them as candidate equivalence conditions for correlated-gate pairs which may first mismatch at time i (set M_i) is effective. In other cases, e.g. power-gated designs, more importantly than when correlated-gate pairs may first *mismatch* is when they may first transition from inequivalent to *equivalent*.

(2) Until a gate toggles for the first time, that gate is effectively constant hence is not useful as an equivalence condition for any correlated-gate pair. Similarly, until a pair of correlated gates mismatch for the first time, there is no need to attempt to learn invariants over them. Doing so would entail an unconditional equivalence invariant – or tautology – which may be handled more efficiently as per Section III-C.

(3) Often, many correlated-gate pairs share the same equivalence conditions. For example, each state element of a given pipeline stage in Figure 2 has the same Valid_i as an equivalence condition. It thus is often useful to assess the invalidity of a large set of candidate equivalence conditions for a small subset of correlated-gate pairs, e.g., several representatives identified at a given time-frame in Step 3 of the algorithm of Figure 5. Only those which appear valid for this subset will be considered for the remaining correlated-gate pairs.

(4) Structural analysis may be used to further prune candidates in cases, e.g., to only consider gates with substantial fanin overlap between a correlated-gate pair as equivalence condition candidates thereof. Such a technique often works well for clock-gating and power-gating verification, as the equivalence condition for a given state-element pair is often directly used to clock one of the two state elements. However, for more general sequential ODC-based optimizations, such structural prunings may fail to capture adequate conditions which may only be present in logic which flows around the redesigned subcircuits and/or be present at the testbench level alone. Nonetheless, in applicable cases such structural pruning may dramatically improve runtime: either limiting equivalence condition candidates to overlap with the fanin cone of the correlated-gate pair, or limiting conditions to be gates in the testbench logic.

(5) While we have not yet needed to rely upon such techniques, it is possible to use known invariant-reduction methods to minimize the cardinality of the candidate invariant set either in a semantically lossless or lossy manner. For example, *transitive reductions* may be used to reduce the number of implication-based invariants with no loss in their semantic power [14]. Just as invariants $(a \rightarrow b)$ and $(b \rightarrow c)$ subsume invariant $(a \rightarrow c)$, one may apply similar subsumption rules for conditional equivalence invariants: if $(a \rightarrow b)$, then $\big(a \rightarrow (m_i \equiv m_i')\big)$ subsumes $\big(b \rightarrow (m_i \equiv m_i')\big)$. Additionally, one may attempt lossy candidate invariant pruning using techniques such as ranking the relative constraining power of the candidates and retaining only a subset of greatest strength [16].

C. Incorporating Unconditional Equivalence Invariants

While the primary goal of our CSEC solution is to establish invariants over correlated-gate pairs which are conditionally equivalent (and thus conditionally *inequivalent*), in practice the netlists being equivalence checked may also contain a subset of correlated-gate pairs which are *unconditionally* equivalent. Such unconditional equivalence often must be considered to ensure inductiveness of the overall CSEC problem.

Merely using traditional SEC algorithms to prove-then-merge unconditionally-equivalent (or constant) gates, prior to application of our CSEC invariant-generation framework, may partially capture the unconditional equivalences. However, this approach is often insufficient since the presence of the conditionally-equivalent gates, without their corresponding conditional-equivalence invariants, renders the proof of unconditional equivalence intractable.

The most effective way that we have found to intermix the conditionally- and unconditionally-equivalent gate demonstration solutions is to nest the proposed conditional-equivalence invariant generation algorithm inside a traditional SEC framework using *speculative reduction* [5], [6]. Specifically, one may first postulate a set of candidate unconditional equivalences using a traditional SEC framework. Then, instead of directly proving those candidates, one may speculatively reduce the netlist such that fanout references to the candidate unconditional equivalences reflect a merge, thereby simplifying the overall set of proof obligations – while retaining a proof obligation to validate that the speculative merge candidates truly are equivalent. This speculatively-reduced model may be used as the basis of our CSEC framework, which will be used to prove the overall CSEC objectives in addition to the unconditional speculative-reduction proof obligations.

A comparable solution is to first generate candidate conditional-equivalence invariants, then to use a traditional SEC framework to prove those invariants in conjunction to any unconditional equivalences. This approach tends to be somewhat less efficient, however, as the CSEC framework needs to operate on a larger netlist, possibly managing more semantically-equivalent invariants.

D. Auxiliary Invariants

In addition to conditional and unconditional equivalence invariants, it is occasionally necessary to establish other types of invariants to ensure an overall inductive CSEC problem. For example, consider the CSEC testbench in Figure 2. This testbench includes two copies of the netlist of Figure 1 – N with an unconstrained Valid_1 to enable clock-gating, and N' with clock gating disabled. Additionally, some auxiliary testbench logic is often used to synthesize the equivalence condition under which output equivalence will be checked. For simplicity, in Figure 2 this condition is simply $Valid_4$ directly from N. However, due to the risk that the clock-gated design is improperly implemented, it is often desirable to use a correct-by-construction testbench-level equivalence condition.

It is sometimes necessary to derive invariants that establish a correlation between such testbench-level equivalence condition logic, and logic within the netlists being equivalence checked which indicates inactivity or *don't care* conditions. For example, in Figure 2, these invariants would simply correlate stages of the *valid* pipeline created at the testbench level as being equivalent to the corresponding $Valid_i$ signals in the clock-gated netlist. If such invariants are missing, k-induction may fail, confused by possible inductive starting states where the testbench logic indicates that output equivalence is to be checked, yet the gated netlist appears inactive hence conditional equivalence invariants do not apply.

When necessary, such invariants often may be established as a byproduct of an industrial-strength verification framework: either through lighter-weight structural simplifications (e.g., redundancy removal) commonly employed to reduce the size and complexity of a verification problem prior to invoking any costly decision procedures, or through the use of a coupled SEC framework to identify unconditional equivalences prior or subsequent to establishing conditional equivalence invariants as discussed in Section III-C.

The proposed invariant generation framework is generally incomplete in its goal of enabling an overall inductive or interpolation-amenable CSEC solution. While we have not encountered cases where such a framework was inadequate, we believe that – even if more elaborate invariants of this type were to be necessary – they would be straight-forwardly enumeratable by the engineer who set up the CSEC testbench given the need to have a basic understanding of how to specify the equivalence condition mapping C'' thereof. Additionally, we note that the proposed framework is readily incorporatable with other invariant generation and proof techniques. As our experiments demonstrate, we have found the proposed framework extremely powerful in enabling the automated solution of classes of difficult CSEC problems which we otherwise were not able to solve using any algorithms available to us.

E. Speculative Reduction over Conditional Equivalences

The concept of speculative reduction may be extended from unconditional equivalences – a fanout-merge – to conditional equivalences by injecting a multiplexor. Specifically, given $(m_i, m_i') \in M''$, one may replace fanout references to m_i' by the function *if* $(\bigvee_{g_i \in E(m_i)} g_i)$ *then* (m_i) *else* (m_i'). However, unlike speculative reduction for unconditional merges which is guaranteed not to increase the size of the resulting netlist, speculative reduction for conditional merges is almost guaranteed to increase netlist size. We thus have not yet experimented with the use of this technique, though postulate that it may benefit various algorithms in structurally reflecting the ODC condition implicit in the set of conditional equivalence invariants.

IV. Experimental Results

In this section we provide experimental results to illustrate the benefits of our conditional equivalence invariant generation techniques. The set of CSEC problems that we have encountered to date is admittedly somewhat small: there are no public CSEC benchmarks available to our knowledge (most are direct SEC problems), and the majority of industrial problems also fall into the CEC or SEC category: even if they contain sequential ODC-style optimizations, those optimizations may often be verified using a highly-tuned multi-algorithm SEC toolset to cope with any noninductive subcircuits (e.g., [5], [6]). We nonetheless have found this invariant generation framework useful to enable efficient inductive solutions to numerous of these type of problems, using whichever heuristics discussed in Sections III-A and III-B apply to the particular problem under consideration.

There are however numerous CSEC problem domains where even the best-tuned SEC solution simply cannot scale, such as power-gating and globally-optimal clock-gating. This work has been motivated by the existence of such difficult CSEC examples which we have been unable to solve without substantial manual decomposition or abstraction, and the knowledge that such problem domains will increase in prevalence as the demand for low-power devices continues to rise. Furthermore, as the sophistication of synthesis tools continues to grow, their ability to leverage more intricate ODC conditions will also continue to grow; equivalence checking tends to be less scalable than synthesis itself, since the algorithms in the former generally need to reason about a design twice as large as the latter. We thus focus our experimental discussion on one representative example from each of these categories of difficult CSEC problems that otherwise were unsolvable using existing algorithms. In these examples, the gates in M'' are restricted to state elements. Our invariant generation framework has been implemented in the IBM internal verification tool *SixthSense* [26], [5], [10], [6], [27].

A. Case Study 1: Clock Gating

This example is a high-performance double-precision floating-point unit (FPU) with fused multiply-add capability. It contains a 53x54 multiplier, adders, shifters, a leading-zero anticipator, and other logic components typically associated with FPUs. It has a pipeline of 12 clock periods due to its high clock frequency. This FPU is designed to operate configurably with or without clock-gating enabled; this testbench validates the conditional equivalence of these two operational modes.

The issuing of an instruction into the FPU pipeline is indicated by the assertion of an *Issue* primary input. Internally, this input is pipelined, and this pipeline is used to indicate whether there is a valid instruction at a particular stage so that its clock is enabled (similar to the Valid_i pipeline of Figure 1). At the testbench level, *Issue*s are also pipelined and a conditional output equivalence check is associated with this testbench-level pipeline.

The complexity of the arithmetic logic, and the high degree of bit-level optimization thereof, make this example particularly challenging. The design itself comprises more than 23k lines of RT-level VHDL, which does not include several synthesized components. This complexity prevents us from performing even bounded model checking of a single instruction through the FPU within 24 hours. The only way that we have been able to solve this CSEC problem in the past, before the automated conditional equivalence invariant generation solution was available, has been through time-consuming manual abstraction of the arithmetic subfunctions between pipeline stages using uninterpreted functions, then brute-force automated abstraction and reachability algorithms.

This CSEC netlist has 21,453 state elements, 542 inputs, and 130,412 And gates. We limited conditional equivalence candidates to be those appearing at the testbench level alone, due to the creation of the above-mentioned correct-by-construction pipeline indicating when output equivalence is to be checked. The goal of the conditional equivalence checking invariant generation framework is thus to identify which stages of the testbench pipeline imply conditional equivalence of which correlated state elements of the two versions of the FPU. There are 254,016 candidate invariants using this approach. Of these, bounded model checking falsified 357, whereas our bit-parallel simulator falsified 242,330: a ratio of 1:679. The remaining 11,329 invariants were proven valid using 3-step induction, from which the overall CSEC objectives became 1-step inductive. The number of gates participating in any conditional equivalence condition set E for these invariants was 16. The resources required for this run are 14,459 seconds on a 2.16 GHz processor, with 2.0 GB peak memory.

B. Case Study 2: Power Gating

Our next case study demonstrates the applicability of our conditional equivalence invariant generation framework to solve difficult power-gating CSEC problems. This testbench is the result of applying the power-gating verification methodology presented in [8]. The design being equivalence checked is a four-port out-of-order execution unit, which can compute loads, stores, adds, subtracts, shifts, and branches on 32-bit data, manipulating a 16-entry register file. This CSEC testbench includes two copies of the execution unit: one with power-gating enabled, and one with power-gating disabled. This design comprises more than 13k lines of RT-level VHDL.

The verification setup of [8] performs SEC under conditions imposed by an *observer*. The observer is a testbench-level circuit that observes the netlists being equivalence checked, and raises a flag when the execution unit is active (i.e., is working on some command). This flag constitutes the equivalence condition gate g''_o under which each output o is equivalence checked. This flag also reflects the violation of environment-assumption violations; g''_o forever deasserts after such a violation, trivializing the equivalence check. This example is an extended version of the one reported in [8], which is substantially more challenging to solve. *All previous attempts to solve this CSEC problem failed, even when given virtually unlimited memory and time limitations, using any available algorithms*. Even the manually-simplified variant of this problem reported in [8] required 91 hours of runtime.

This example used the algorithm of Figure 4, establishing that when not tristated, correlated state-element pairs are equivalent. Additionally, we added candidate invariants checking whether each state element may ever be tristated to further strengthen the induction hypothesis; similar strengthening could be achieved by nesting the invariant-generation framework within a SEC framework as discussed in Section III-C. We used an automated technique to safely learn *verification constraints* [28], correlating to the observer having witnessed assumption violations which forever thereafter entails a trivial pass of the equivalence check. These constraints enhanced the inductivity of the CSEC objectives, as uninteresting behavioral differences between the two designs became unreachable. We additionally leveraged automated phase abstraction [29] to temporally compress the clock period from two to one verification time-step, as well as an automated temporal shifting algorithm to eliminate the reset phase from the testbench [27], to collectively enhance inductiveness of the CSEC objectives.

This CSEC netlist has 807 state elements, 175 inputs, 22,326 And gates, and 417 outputs to be equivalence checked. Of 1,196 candidate invariants, 961 were proven valid using 3-step induction. The number of gates participating in any conditional equivalence condition set E for these invariants was 669. These invariants enabled all outputs to be proven equivalent with 1-step induction, with a total runtime of 149 seconds and 97.2 MB on a 2.16 GHz processor.

V. Conclusion

In this paper we have proposed an eager conditional equivalence invariant generation framework to enable the solution of difficult *conditional sequential equivalence checking* (CSEC) problems. Such problems, which commonly arise due to design optimizations such as clock gating and power gating, render traditional SEC frameworks highly unscalable due to the absence of internal unconditionally-equivalent

points. Nonetheless, often there are numerous internal points which are only inequivalent when such inequivalence is an *observability don't care*. Our invariant generation framework attempts to identify an adequate set of conditional equivalence invariants to render an efficient overall CSEC solution through scalable proof techniques such as induction. Our experiments demonstrate the power of this approach, enabling the efficient automated solution of several classes of problems which we otherwise were unable to solve using any available algorithms.

REFERENCES

[1] A. Kuehlmann and C. A. van Eijk, *Combinational and Sequential Equivalence Checking*. Kluwer Academic Publisher, 2001.

[2] J.-H. Jiang and R. Brayton, "Retiming and resynthesis: A complexity perspective," in *TCAD*, vol. 25, Dec. 2006.

[3] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, Feb. 1998.

[4] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, Nov. 2000.

[5] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[6] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative-reduction based scalable redundancy identification," in *DATE*, Apr. 2009.

[7] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi, *Low Power Methodology Manual*. Springer, 2007.

[8] C. Eisner, A. Nahir, and K. Yorav, "Functional verification of power gated designs by compositional reasoning," in *CAV*, July 2008.

[9] R. Brayton and A. Mishchenko, "Recording synthesis history for sequential verification," in *FMCAD*, Nov. 2008.

[10] J. Baumgartner, H. Mony, and A. Aziz, "Optimal constraint-preserving netlist simplification," in *FMCAD*, Nov. 2008.

[11] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.

[12] K. McMillan, "Interpolation and SAT-based model checking," in *CAV*, July 2003.

[13] G. Cabodi, S. Nocco, and S. Quer, "Strengthening model checking techniques with inductive invariants," *TCAD*, vol. 28, Jan. 2009.

[14] M. Case, A. Mishchenko, and R. K. Brayton, "Inductively finding a reachable state space over-approximation," in *IWLS*, 2006.

[15] K. McMillan, "Don't care computation using k-clause approximation," in *IWLS*, 2005.

[16] M. Case, A. Mishchenko, and R. Brayton, "Cut-based inductive invariant computation," in *IWLS*, 2008.

[17] L. de Moura, H. Rueß, and M. Sorea, "Bounded model checking and induction: From refutation to verification," in *CAV*, July 2003.

[18] M. Case, A. Mishchenko, and R. Brayton, "Automated extraction of inductive invariants to aid model checking," in *FMCAD*, Nov. 2007.

[19] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, March 2005.

[20] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *DAC*, July 2006.

[21] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, March 1999.

[22] M. Turpin, "The dangers of living with an X," in *SNUG*, Oct. 2003.

[23] C. Pixley, "A theory and implementation of sequential hardware equivalence," in *TCAD*, Dec. 1992.

[24] Z. Khasidashvili and Z. Hanna, "SAT-based methods for sequential hardware equivalence verication without synchronization," in *BMC*, 2003.

[25] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *ICCAD*, Nov. 2004.

[26] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[27] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman, "Enhanced verification through temporal decomposition," in *FMCAD*, Nov. 2009.

[28] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer, "Speeding up model checking exploiting explicit and hidden verification constraints," in *DATE*, April 2009.

[29] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification," in *ICCAD*, Nov. 2005.

Verifying Equivalence of Memories Using a First Order Logic Theorem Prover

Zurab Khasidashvili, Mahmoud Kinanah
Intel Israel (74) Ltd.
Haifa 31015, Israel
{zurabk,mkinanah}@iil.intel.com

Andrei Voronkov
Computer Science Department
University of Manchester, UK
{andrei@voronkov.com

***Abstract*—We propose a new method for equivalence checking of RTL and schematic descriptions of memories using translation into first-order logic. Our method is based on a powerful abstraction of memories and address decoders within them. We propose two ways of axiomatizing some of the bit-vector operations, decoders, and memories. The first axiomatization uses an algebra of operations on bit-vectors. The second axiomatization considers a bit-vector as a unary relation and memory as a relation of larger arity. For some designs, including real-life designs, the second axiomatization results in a first-order problem falling into a known decidable fragment of first-order logic and suitable for solving by modern first-order provers. Equivalence of real-life memories can be verified in seconds with our approach.**

I. Introduction

Formal Equivalence Checking [10] is an important part of the logic verification of hardware. Its aim is to check that the specification and implementation models of hardware have an equivalent logic functionality, according to some concept of equivalence. In contemporary design, the specification model is written in a Register-Transfer Level (RTL) hardware description language such as Verilog or System Verilog. The implementation model is a schematic (SCH) model built from transistors. For performing equivalence checking, the specification and implementation models are represented as finite state transition systems [8] and then compared.

Advances in term-level model checking techniques and decision procedures dealing with uninterpreted (or partially interpreted) functions, bit-vectors, equational theories, and extensional theories of arrays [16] enable reasoning about the RTL model at a much higher level of abstraction than the bit-level, where the states and the transition relation are represented as Boolean functions. In the past, it was impossible to apply advanced term-level verification algorithms to the equivalence checking of RTL and schematic models since the schematics extraction tools [7], [21] could identify logic and state elements (e.g., latches) in the circuit but could not identify memory arrays, decoders, and bit-vector operations.

Equivalence checking of memories is one of the most complex tasks in equivalence checking. Recent advances in sequential equivalence checking [11], [2], [4] allow more freedom in memory design in the RTL, in that corresponding memories in RTL and SCH models need not be *state-matching*. That is, the differences between RTL and SCH memory designs can be in the number and placement of state elements, and not only in the implementation of Boolean logic. In the terminology of equivalence checking this means that not every state element in the RTL model needs to be *mapped* to a state element in the SCH model. Still, the equivalence checking method currently employed in RTL-to-SCH equivalence checking tools [11] requires that every memory cell in the two models be mapped, and each pair of mapped memory cells is verified for equivalence; the corresponding outputs in the two memories cannot be verified without using the memory cells as cut-points, to prune the cone of influence to a manageable size.

In Intel's experience, despite the availability of smart algorithms that make it possible to verify the equivalence of several mapped node pairs simultaneously, equivalence checking of memory units takes $2-3$ hours on average. This is due both to the complexity of corresponding model-checking instances (which contain the complex address decoding logic) and due to the very large number of verification pairs. Still, this is only a part of the problem with the current method of memory equivalence checking. Since memory design in the SCH model can differ significantly from the design of the corresponding memory in RTL (for example, a memory in RTL might be implemented as several small memories in the SCH model, and the placement of the rows and columns might be very different), manual work is often required to map the memory cells correctly. If the equivalence checking tool reports a mismatch for a pair of mapped memory cells, debugging may reveal that the mismatch has been caused by an incorrect mapping rather than a bug in the designs. The mapping must then be corrected and verification rerun. Such verification and debugging iterations on a memory unit take, on the average, 1-2 days of effort on the part of a verification engineer. This is a significant overhead that we want to avoid using our new verification approach.

Advanced term-level equivalence checking tools for RTL-to-RTL equivalence checking and ESL-2-RTL equivalence checking avoid bit-blasting of memories by employing smart abstraction techniques for modeling memories with far fewer bits than the number of memory cells in them (here ESL stands for Electronic System Level design) [17], [13]. The ESL-to-RTL equivalence checking tool requires mapping as the user input, while in [17] the mapping of memory cells (or rather, of array locations) is implicit and is the identity function.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

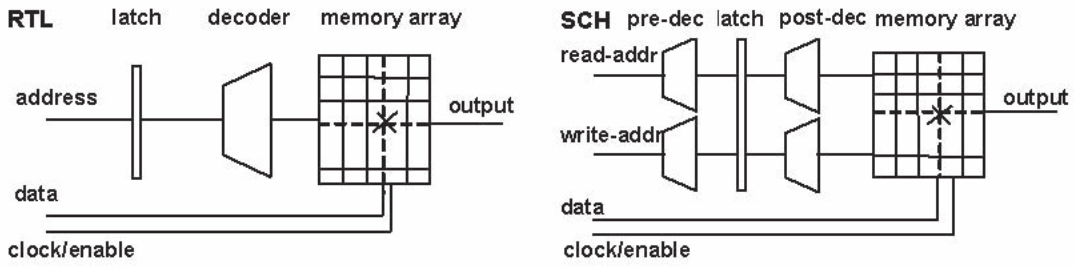


Fig. 1. RTL and SCH memories.

We propose an RTL-2-SCH equivalence checking algorithm for memories that does not require mapping memory cells and allows for modular verification and debugging. Our method is based on a powerful abstraction of address decoders. Properties of decoders and some bit-vector operations are represented in First-Order Logic (FOL), and we reduce the equivalence checking of memories to solving quantified formulas in a known decidable sub-class of FOL. Previous work on verifying memories using FOL is based solely on the axiomatization of arrays, while decoder axiomatization has never been considered. In addition, some previous work on using quantified FOL formulas in the verification of memories encode the equivalence checking problem into an undecidable class of FOL, and dedicated heuristics have been developed to solve such formulas efficiently [1].

With an alternative encoding, after abstracting the address decoders, the equivalence checking problem can be solved efficiently by SMT solvers that can reason with extensional axiomatization of arrays [6]. Our encoding of memories and bit-vectors is conceptually different from known SMT- and FOL-based techniques, as well as from the smart memory abstraction techniques seen in [9], [17], [13], [5]. It results in formulas in the decidable fragment of first-order logic known as EPR formulas or the Bernais-Schönfinkel class. Experimental results performed using the Vampire [22] FOL theorem prover and other tools show that representative real-life memories can be verified in a few seconds using our method.

The paper is organized as follows. In the next section we review RTL and schematic memory design, focusing on design differences caused by timing, power, and layout optimizations in the SCH designs. In Section III we present an axiomatization of address decoders. In Sections IV and V we discuss a representative real-life equivalence checking problem and propose two ways of representing bit-vectors in FOL. The better of the two, as shown by the experimental results presented in Section VI, gives problems in a decidable fragment of FOL. Conclusions appear in Section VII.

II. Memory design and implementation

In contemporary chip design, an estimated 20% of all functional units are memories of various kinds and in schematics they occupy 50% of the chip area. An average size memory contains tens of thousands of memory cells. There is a need to optimize the implementation of memories for high performance, low power consumption, and efficient layout.

Sequential (i.e., non-state-matching) equivalence checking [10] enables freedom in memory design in RTL, in that the RTL designer can focus only on a concise description of the functionality of a memory unit, leaving all optimization to the circuit designer. This abstraction of the details of memory implementation in the RTL makes for a cleaner and faster RTL design, prevents low-level design bugs from entering the RTL, and makes the functional validation of RTL simpler. It is the task of equivalence checking to prove that the implementation is equivalent to the RTL description.

In RTL, memory write and read operations can be specified, for example, as follows:

```
memory[address[7:0]][63:0]=write_data[63:0]
                                          (write)
read_data[63:0]=memory[address[7:0]] (read)
```

As can be seen in this example, address decoding is implicit for both write and read operations, and the same signal is used for specifying both the read and the write addresses; the addresses are in encoded form – 8 bits of the address define 2^8 physical addresses. Using the same address signal for both read and write operations is safe because the two operations cannot be performed simultaneously. In the implementation model (Figure 1), the read and write decoders may be implemented separately, and the decoding logic might be intermingled with read/write enabling logic. Furthermore, for large memories, because the decoding logic becomes complex, decoding happens in stages separated by latches, whereas in the specification model the latch describing the timing of the decoding logic is placed before the (implicit) decoder (Figure 1). Because of the different placement of the latches, the two designs are not state-matching.

Physical and layout optimizations often require a different organization of memories in the schematic model. Often the memory array is split into two or four chunks, and the rows are grouped differently – odd and even rows being placed in separate chunks. Then the decoder and the memory appear as in Figure 2.

Memories may allow write and read operations via several ports, as in Figure 3. There are several variations: write/read via a certain port might be possible to/from an entire row or only parts of it (say to the MSB half or the LSB half).

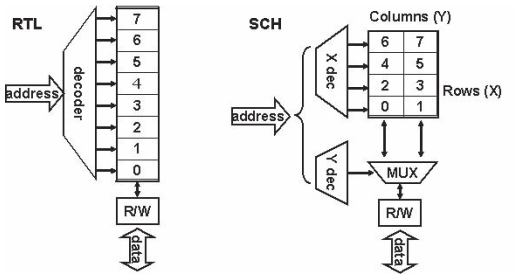


Fig. 2. Decoding and layout styles.

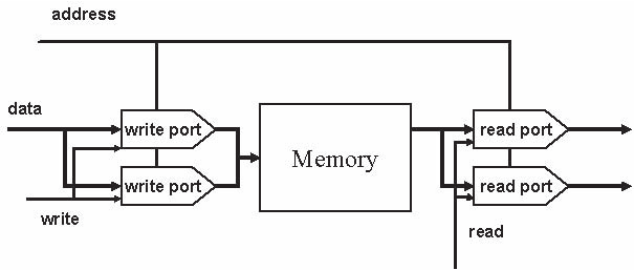


Fig. 3. Memory with two read and write ports.

A previous work on identifying memory cells, in Intel's schematic extraction tool, is reported in [21]. The memory identification capability that was developed there enabled an improved modeling of latches that serve as memory cells in special types of memories. That work does not support the abstraction of a collection of latches into memories, or the identification of decoders. Our work builds on [21]. To enable equivalence checking of RTL and schematic memories at a higher level of abstraction, we have developed a capability for identifying the memory arrays and the address decoders in memory units. In the process of identifying the decoders, the extraction tool retimes the latches inserted between the various stages of decoding before the decoding logic (Figure 1) and merges several stages of decoding into one decoder and several chunks of memory into one memory. It also separates the read and write address decoding logic from the read and write enabling logic.

III. Representing memories and decoders

McCarthy's *consistency* properties of arrays are often used in the literature for verifying memories and arrays in general; here i and j are decoded addresses (or indices) of the memory mem; $mem(i)$ denotes the element of mem at address i, and $mem\{i \leftarrow e\}$ denotes the memory obtained from mem by writing element e at address j in mem:

$$mem\{i \leftarrow e\}(i) = e;$$
$$mem\{i \leftarrow e\}(j) = mem(j), \quad \text{if } j \neq i.$$

We would like the array consistency properties to be valid also when i and j denote encoded addresses; an n-bit encoded address defines 2^n decoded addresses. To this end, our representation needs to deal explicitly with decoding functions for the read and write operations.

In the specification model, we have an implicit decoder, denoted by $decs$, which is (implicitly) used during both write and read operations. We need to assume that $decs$ is one-to-one: encoded addresses A and B are equal if the corresponding decoded addresses are equal. We call this property the *decoding correctness* property:

$$\forall A \forall B((decs(A) = decs(B)) \rightarrow A = B).$$

In the implementation model, we have a write decoder $wdeci$ and a read decoder $rdeci$. Just like for $decs$, we need to assume that $wdeci$ and $rdeci$ are one-to-one. Furthermore, in order for the SCH memory to function correctly, $wdeci$ and $rdeci$ must be the same functions. We call this latter property the *read-write decoding consistency* property:

$$\forall A \forall B((rdeci(A) = rdeci(B)) \rightarrow A = B);$$
$$\forall A \forall B((wdeci(A) = wdeci(B)) \rightarrow A = B);$$
$$\forall A : (rdeci(A) = wdeci(A)).$$

The decoding correctness property in the specification model is correct by construction, based on how the RTL compiler generates the decoders for register files. The decoding correctness property in the implementation model and the read-write consistency property in the implementation model are checked by the circuit extraction tool as part of identifying the read and write decoders. The tool checks that any address can activate only one row for read and for write, that the same address activates the same row both for read and for write, and that different addresses cannot activate the same row. Thus a bug in the decoders (violating a decoding property) can be identified during the schematic extraction process, and can be reported to the user before the equivalence checking tool is run.

Memories in the schematic model might contain redundant (i.e., unaccessible) columns and/or rows. During decoder recognition, the schematics extraction tool identifies the redundant columns and rows and thereby makes it possible to ignore them in the equivalence checking completely automatically. Redundant rows can be safely removed from the memories (for the purpose of equivalence checking), while redundant columns are "hidden" from the equivalence checking in the encoding of equivalence checking problem to FOL provers.

After the decoding properties have been checked, the equivalence checking problem of memories with encoded addresses and explicit decoders is reduced to the significantly simpler problem of equivalence checking of memories with encoded addresses and implicit decoders. Indeed, since the choice of an implementation of a correct decoder does not affect the input-output behavior of a memory unit, we can assume that the decoders $decs$, $wdeci$ and $rdeci$ in the two memory models are the same functions, and we abstract them away and treat them implicitly.

The concept of equivalence we are working with is *post-reboot observational equivalence* [12]. For proving the equivalence of RTL and SCH memories, we define an inductive invariant that states that the corresponding cells and the corresponding outputs in the two memories satisfy a certain relation, e.g., have the same or inverse values. Below we will

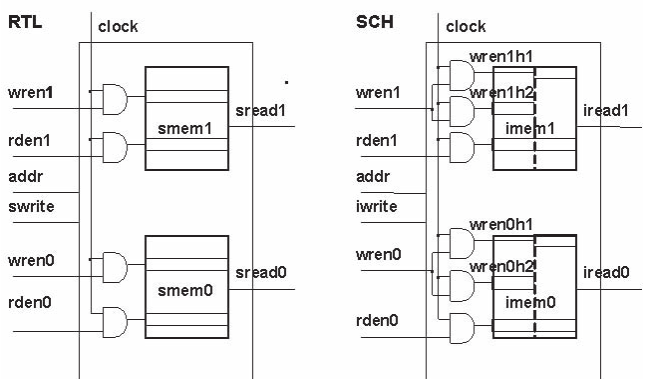


Fig. 4. Specification and Implementation memories

consider these two last-mentioned, most widespread, memory correspondence relations for the RTL and SCH memories.[1] To define the memory-correspondence invariant, we do not need a mapping between the memory cells of the RTL and the original SCH model. Owing to decoder abstraction, we only need to map the address signals in the two models and specify the polarity of the mapping. We use the information about redundant columns in the SCH model when building the invariant formula - the invariant formula does not impose any requirements on the bits corresponding to the redundant columns.

We assume that the memories can be brought to initial states that satisfy the invariant. Indeed, it is easy to design an input sequence that can set every cell in the memory to the same value (to constant F or constant T); for many memories this can be done using the set and reset signals, in one clock cycle. In such states, the memory-correspondence part of the invariant can be checked without knowing the actual correspondence between the memory cells. We use induction to prove the invariant. This is described in Sections IV and V.

Strictly speaking, there is no need to actually build (or prove the existence of) an initializing sequence that brings the memories into initial states satisfying the invariant. The compositional equivalence checking theory in [12] reduces post-reboot observational equivalence checking to checking that the reboot sequence satisfies certain initialization requirements and that the corresponding modules in the decomposition have a pair of states equivalent under valid, possibly temporal, assumptions. In the context of the equivalence checking of memories, examples of useful assumptions include assumptions ensuring that the read and write enables of a memory cannot assert in the same phase; such assumptions are valid in the designs but they need to be added to the equivalence checking instance when the read and write enable signals are in the interface of the memory modules that are compared.

IV. Algebraic approach

In this approach we consider bit-vectors as scalar values and memories as functions. In a way, this representation is similar to some formalizations of memory in software verification and static analysis, where the memory is a mapping from addresses to values [16]. We thus use an axiomatization of arrays for reasoning about memory. However, to verify real-life designs, we need to axiomatize some operations on bit-vectors; we illustrate this approach on an example shown in Figure 4. We will refer to this example as our *running example*. The example consists of the specification and the implementation memories; they are representative memory models obtained by analyzing tens of real-life specification and implementation memory units in a recent Intel micro-processor project.

The specification memory consists of two parts $\texttt{smem}_1$ and $\texttt{smem}_0$. There are two bit-vectors: `addr` is used to pass the memory address for both read and write operations and `swrite` is used to denote the data for the write operations. There are four bits for enabling write and read operations on the two memories: $\texttt{wren}_1$, $\texttt{rden}_1$, $\texttt{wren}_0$ and $\texttt{rden}_0$. Finally, there are two bit-vectors, $\texttt{sread}_1$ and $\texttt{sread}_0$, used for writing the result of read operations. The implementation is similar to the specification; however accessing the memories is very different. The write bit-vector data is split into two parts. Each of the parts is written to the corresponding part of the implementation memory, so the implementation memory is shown split into two parts. Beyond the boundaries of the implementation memory unit the data is bitwise negated before being written to the implementation memory and after being read from it, so the data in the implementation memory is the negation of the data in the specification memory.

To express this example using the algebraic approach we need to represent all the bit-vectors and memories used in it. To this end we will use the constant and function names shown in Figure 5, where $i = 0, 1$. To represent the transition relation, we will need the "next-state" versions of these variables. They will always be represented by adding a prime symbol $'$ to these variables. For example, $\texttt{iwrite}'$ refers to the next state version of `iwrite`. There will be several propositional input variables. Using the fact that they have the same values in the specification and the implementation, we will not introduce their specification and implementation versions. Instead, they will be shared between the specification and the implementation.

We start with the specification of state correspondence (the invariant) between the specification and the implementation. The invariant implies the *equivalence* [10] of the states, meaning that the corresponding output values in the corresponding states are equal and remain equal along any transition path. We define the correspondence as follows:

$$\texttt{iread}_0 = \mathit{neg}(\texttt{sread}_0) \wedge \texttt{iread}_1 = \mathit{neg}(\texttt{sread}_1) \wedge$$
$$\forall A : (\texttt{imem}_0(A) = \mathit{neg}(\texttt{smem}_0(A))) \wedge$$
$$\forall A : (\texttt{imem}_1(A) = \mathit{neg}(\texttt{smem}_1(A))).$$

[1] Our method can be extended to cover more complex memory correspondence relations, the cost being that the encoding to FOL solvers may yield formulas outside the EPR fragment.

constant	explanation
$\texttt{iread}_i$	implementation read data i
iwrite	implementation write data
$\texttt{sread}_i$	specification read data i
swrite	specification write data
function	explanation
$\texttt{imem}_i$	implementation memory i
$\texttt{smem}_i$	specification memory i

Fig. 5. Symbols used in specifying memory designs

We will use a simple induction for proving behavioral equivalence of the two designs, so our goal is to prove the equivalence of the next states:

$$\texttt{iread}'_0 = neg(\texttt{sread}'_0) \wedge \texttt{iread}'_1 = neg(\texttt{sread}'_1) \wedge$$
$$\forall A : (\texttt{imem}'_0(A) = neg(\texttt{smem}'_0(A))) \wedge$$
$$\forall A : (\texttt{imem}'_1(A) = neg(\texttt{smem}'_1(A))).$$

Note that for functions over bit-vectors we have to introduce functions in the design; for example, neg represents a bitwise negation. This comes at a price: we may also have to introduce additional formulas expressing properties of these functions. When the number of functions used in the design is large and the combinatorics is complex these additional formulas may slow down proving equivalence considerably. They cannot, however, be omitted in general.

The RTL description of the circuit contains propositional variables clock, $\texttt{wren}_i$ and $\texttt{rden}_i$ which denote the current clock value and conditions for enabling write and read operations, respectively. We add the following environment assumptions (which are true by design) about these operations (below, in all formulas, $i = 0, 1$):

$$\neg\texttt{wren}_i \vee \neg\texttt{rden}_i.$$

The next formula relates the implementation and specification values of the write data:

$$\texttt{iwrite} = neg(\texttt{swrite}). \tag{1}$$

The RTL description contains definitions of the transition relation for all bit-vectors and memories. These definitions can be automatically translated to the following formulas:

$$\begin{array}{l}
\forall A(\texttt{clock} \wedge \texttt{wren}_i \wedge A = \texttt{addr} \rightarrow \texttt{smem}'_i(A) = \texttt{swrite}); \\
\forall A(\neg(\texttt{clock} \wedge \texttt{wren}_i \wedge A = \texttt{addr}) \rightarrow \texttt{smem}'_i(A) = \texttt{smem}_i(A)); \\
\texttt{clock} \wedge \texttt{rden}_i \rightarrow \texttt{sread}'_i = \texttt{smem}_i(\texttt{addr}); \\
\neg(\texttt{clock} \wedge \texttt{rden}_i) \rightarrow \texttt{sread}'_i = \texttt{sread}_i; \\
\texttt{clock} \wedge \texttt{rden}_i \rightarrow \texttt{iread}'_i = \texttt{imem}_i(\texttt{addr}); \\
\neg(\texttt{clock} \wedge \texttt{rden}_i) \rightarrow \texttt{iread}'_i = \texttt{iread}_i.
\end{array} \tag{2}$$

For example, the first formula means that, whenever the write operation is enabled for the specification memory i and the value of addr is A, then in the next state the value of $\texttt{smem}_i(A)$ is equal to the value of the write data bit-vector swrite. Likewise, the second formula means that, when the write operation is not enabled for the specification memory i or A is different from the value of addr, the specification memory content at A is not changed by the transition.

The next group of (propositional) formulas comes from the RTL description that contains propositional variables $\texttt{wren}_{1,h1}$ and $\texttt{wren}_{1,h2}$ for enabling writing to the two parts of the memories:

$$\begin{array}{l}
\texttt{wren}_{i,h1} \leftrightarrow \texttt{wren}_i \wedge \texttt{clock}; \\
\texttt{wren}_{i,h2} \leftrightarrow \texttt{wren}_i \wedge \texttt{clock}.
\end{array}$$

An interesting part of the specification is the part that splits a bit-vector having k bits into a pair consisting of the first r bits and the remaining bits. In our example, we assume that all bit-vectors have 71 bits, numbered from 0, and we split the vector into a 36 and a 35 bit-vector. To model this, we introduce bit-extraction functions. A bit-extraction function has the form $\texttt{bit}_{l,m,k}$ and denotes the bit-vector obtained by extracting the bits between l and m from a bit-vector of length k. There is a potentially unlimited number of bit-extraction functions; the functions needed to model a particular design can be found automatically from the corresponding RTL or SCH description. In our case we need two such functions. For the bit-extraction functions we need a formula expressing that, if both parts of two bit-vectors obtained by such functions coincide, then the bit-vectors themselves coincide too.

$$\begin{array}{l}
\texttt{bit}_{0,35,71}(X) = \texttt{bit}_{0,35,71}(Y) \wedge \\
\texttt{bit}_{36,70,71}(X) = \texttt{bit}_{36,70,71}(Y) \rightarrow X = Y.
\end{array}$$

In addition, for all bitwise bit-vector functions used in the design (such as neg in our example) we should add formulas expressing that they commute with the bit-extraction functions:

$$\begin{array}{l}
\texttt{bit}_{0,35,71}(neg(X)) = neg(\texttt{bit}_{0,35,71}(X)); \\
\texttt{bit}_{36,70,71}(neg(X)) = neg(\texttt{bit}_{36,70,71}(X)).
\end{array}$$

The rest of the specification is a description of the next state function for the implementation memory ($i = 0, 1$):

$$\begin{array}{l}
\forall A(\texttt{wren}_{i,h1} \wedge A = \texttt{addr} \rightarrow \\
\quad \texttt{bit}_{0,35,71}(\texttt{imem}'_i(A)) = \texttt{bit}_{0,35,71}(\texttt{iwrite})); \\
\forall A(\neg(\texttt{wren}_{i,h1} \wedge A = \texttt{addr}) \rightarrow \\
\quad \texttt{bit}_{0,35,71}(\texttt{imem}'_i(A)) = \texttt{bit}_{0,35,71}(\texttt{imem}_i(A))); \\
\forall A(\texttt{wren}_{i,h2} \wedge A = \texttt{addr} \rightarrow \\
\quad \texttt{bit}_{36,70,71}(\texttt{imem}'_i(A)) = \texttt{bit}_{36,70,71}(\texttt{iwrite})); \\
\forall A(\neg(\texttt{wren}_{0,h2} \wedge A = \texttt{addr}) \rightarrow \\
\quad \texttt{bit}_{36,70,71}(\texttt{imem}'_i(A)) = \texttt{bit}_{36,70,71}(\texttt{imem}_i(A))).
\end{array}$$

The algebraic approach is very flexible. However, using a large number of bit-vector operations (though unlikely in memory designs) may result in adding a number of complex formulas and making it hard for a prover to prove the equivalence. Moreover, there is no guarantee that the first-order axiomatization of bit-vector operations is complete. For example, even in the case of negation it may turn out that we need the axiom $neg(neg(B)) = B$. Giving a larger number of such axioms may result in a problem too difficult for modern theorem provers. Too few axioms may be insufficient to prove the desired property. It will be interesting in the future to modify our translation to work with SMT solvers that can work with quantifiers: this may help in solving the incompleteness problem arising from the axiomatization of bit-vector operations.

Another problem with the algebraic approach is that designs may contain shift operations on bit-vectors or assignments such as $\texttt{bva}[3:2] = \texttt{bvb}[2:1]$, where bva and bvb are, say, bit-vectors with width 32. There is no computationally good way to handle these operations in first-order logic so that bit-blasting is avoided: to deal with the assignment $\texttt{bva}[3:2] = \texttt{bvb}[2:1]$, we introduce axioms $\texttt{bit}_{3,3,32}(\texttt{bva}) = \texttt{bit}_{2,2,32}(\texttt{bvb})$ and $\texttt{bit}_{2,2,32}(\texttt{bva}) = \texttt{bit}_{1,1,32}(\texttt{bvb})$. Combining first-order logic with arithmetic as in [15] may solve the problem, but implementing such a combination is a very hard problem.

V. Relational approach

In this approach we consider bit-vectors as unary relations. This has two advantages over the algebraic approach. First, relations are boolean-valued functions, so they capture the semantics of bit-vectors better than arbitrary arrays. Second, many bit-vector operations can be easily (and without lack of completeness) represented by axioms over the corresponding relations.

In this section we illustrate this approach on our running example and show that it can yield problems belonging to the decidable fragment of first-order logic known as the EPR fragment [20], [19]. We can obtain a problem not in EPR if we use arithmetical operations on addresses which are implemented differently in the specification and implementation.

In the relational approach we represent any bit-vector b as a relation on integers. Thus, for every integer k, we have that $b(k)$ is true if and only if the kth bit of b is 1. If b does not have the kth bit, we still assume that $b(k)$ is either true or false. Such a representation of bit-vectors is a powerful abstraction, since, instead of considering a bit-vector a mapping from a finite range of integers to booleans, we consider it as a mapping from *all* integers to booleans. It is precisely due to this abstraction that we avoid bit-blasting and make it possible the use first-order theorem provers. Memory now becomes a binary relation: the first argument denotes an address and the second a bit. For example, $\texttt{imem}_0(a, k)$ means the value of the k-th bit of the element at the address a in $\texttt{imem}_0$.

Let us now consider the encoding of our running example using the relational approach and compare it with the algebraic approach. First, we define the correspondence between the specification and the implementation as follows.

$$\begin{array}{l}\forall B(\texttt{iread}_0(B) \leftrightarrow \neg\texttt{sread}_0(B)) \wedge \\ \forall B(\texttt{iread}_1(B) \leftrightarrow \neg\texttt{sread}_1(B)) \wedge \\ \forall A \forall B(\texttt{imem}_1(A,B) \leftrightarrow \neg\texttt{smem}_1(A,B)) \wedge \\ \forall A \forall B(\texttt{imem}_0(A,B) \leftrightarrow \neg\texttt{smem}_0(A,B))).\end{array}$$

Note that, unlike the algebraic approach where we simply wrote that two bit-vectors are equal, we write that they are bitwise equal. The correspondence of the next-state memories is expressed in a similar way.

We will now describe the rest of the translation of the RTL and SCH designs for the running example using the relational approach. We will omit the (propositional) parts of the translation that are the same as in the algebraic approach.

What makes the relational approach simpler for theorem provers is that there is no need to define bitwise logic operations on bit-vectors and to add axioms describing their relationships, or to add axioms describing that the logic operators commute with bit-extraction functions, since we can simply use logical connectives to represent them. This is illustrated by the specification of the input write data correspondence (the mapping):

$$\forall B(\texttt{iwrite}(B) \leftrightarrow \neg\texttt{swrite}(B)).$$

In formula (1) used in the algebraic approach we had to introduce the bitwise negation function, which requires also adding properties of this function and can potentially yield incompleteness. In the relational approach, the semantics of the bitwise negation is precisely captured by applying negation to $\texttt{swrite}(B)$, so we do not need any additional axioms.

Let us now define the transition relation for the specification memory:

$$\begin{array}{l}\forall A(\texttt{clock} \wedge \texttt{wren}_i \wedge A = \texttt{addr} \rightarrow \\ \qquad \forall B(\texttt{smem}'_i(A,B) \leftrightarrow \texttt{swrite}(B))); \\ \forall A(\neg(\texttt{clock} \wedge \texttt{wren}_i \wedge A = \texttt{addr}) \rightarrow \\ \qquad \forall B(\texttt{smem}'_i(A,B) \leftrightarrow \texttt{smem}_i(A,B))).\end{array}$$

As compared with the corresponding definition (2) in the algebraic approach, memories are now considered as binary functions. However, the first argument of this function ranges over addresses, not over bits. In general, to be able to use the relational approach, one should identify bit-vectors in the design used as addresses. At the end of this section we will make a few remarks about problems arising out of having addresses as arguments to predicates.

The definitions of the read operations for the specification memory are as follows.

$$\begin{array}{l}\texttt{clock} \wedge \texttt{rden}_i \rightarrow \forall B(\texttt{sread}'_i(B) \leftrightarrow \texttt{smem}_i(\texttt{addr}, B)); \\ \neg(\texttt{clock} \wedge \texttt{rden}_i) \rightarrow \forall B(\texttt{sread}'_i(B) \leftrightarrow \texttt{sread}_i(B)).\end{array}$$

The definitions of the read operations for the implementation memory are similar; one should only replace $\texttt{sread}_i$ and $\texttt{smem}_i$ by $\texttt{iread}_i$ and $\texttt{imem}_i$.

Splitting bit-vectors into parts can be done by introducing a predicate true on bits belonging to the LSB part. Where we have many bit-vector splits on different numbers of bits one needs more predicates denoting subsets of bits. We illustrate how the splitting is handled on our running example. We introduce the predicate $\texttt{less}_{36}$, intended to hold on bits with numbers strictly less than 36. Note that we use an abstraction of bit-vectors in which bit-vectors are of a potentially unlimited size, so we give no definitions of this predicate. However, we use the property that a bit B has as index of 36 or more if and only if it satisfies $\neg\texttt{less}_{36}$.

$$\begin{array}{l}\forall A(\texttt{wren}_{i,h1} \wedge A = \texttt{addr} \rightarrow \\ \qquad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'_i(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ \forall A(\neg(\texttt{wren}_{i,h1} \wedge A = \texttt{addr}) \rightarrow \\ \qquad \forall B(\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'_i(A,B) \leftrightarrow \texttt{imem}_i(A,B)))); \\ \forall A(\texttt{wren}_{i,h2} \wedge A = \texttt{addr} \rightarrow \\ \qquad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'_i(A,B) \leftrightarrow \texttt{iwrite}(B)))); \\ \forall A(\neg(\texttt{wren}_{i,h2} \wedge A = \texttt{addr}) \rightarrow \\ \qquad \forall B(\neg\texttt{less}_{36}(B) \rightarrow (\texttt{imem}'_i(A,B) \leftrightarrow \texttt{imem}_i(A,B)))).\end{array}$$

translation type	time (sec)	memory (MBytes)	generating inferences	simplifying inferences	result
Test 1: algebraic	5.7	52.2	189,388	194,474	unsatisfiable
relational	0.1	19.4	7,896	10,507	unsatisfiable
Test 2: algebraic	1025.5	286.9	55,663,356	49,702,181	unknown
relational	0.1	18.5	5,824	7,093	satisfiable
Test 3: algebraic	4.4	35.4	167,702	173,345	unsatisfiable
relational	0.1	19.4	11,767	15,107	unsatisfiable
Test 4: algebraic	938.2	286.9	20,861,220	22,070,968	unknown
relational	0.0	18.5	3,052	4,114	satisfiable

Fig. 6. Statistics of Vampire runs on the algebraic and relational translations.

This completes the relational translation of the running example.

Another interesting feature of the relational translation for the running example is that it falls into a well-known decidable class of predicate logic known as EPR formulas. Namely, it is the class of CNF formulas having no function symbols. This fragment is decidable and there are several theorem provers that implement decision procedures for this class.

The trick that allowed us to avoid bit-blasting in the relational translation was the use of the address bit-vector as a constant in the language. The idea is that, as far as the memory addresses are concerned, all we care about is whether they are equal or not. If we use sophisticated algorithms performing, say, non-trivial arithmetic on addresses which are different in the specification and implementation memory, we may be unable to trace when the addresses used in the specification and the implementation memory are the same and when they are not. To handle such examples we can perform the following trick. Suppose that a_{spec} and a_{imp} are constants denoting addresses in the specification and the implementation memory, respectively. We can also add unary predicates p_{spec} and p_{imp} denoting these addresses and add the formula $a_{spec} = a_{imp} \leftrightarrow \forall B(p_{spec}(B) \leftrightarrow p_{imp}(B))$.

The standard CNF translation of this formula is a set of clauses still belonging to the EPR class. The advantage of adding it is that we can now prove that two addresses are equal, that is, $a_{spec} = a_{imp}$, by proving that they are bit-wise equal, that is, $\forall B(p_{spec}(B) \leftrightarrow p_{imp}(B))$.

We deal with shifts and assignments of the form $\texttt{bva}[3:2] = \texttt{bvb}[2:1]$ as follows. We introduce constants $bitInd_1$, $bitInd_2$, $bitInd_3$ to denote bit-positions 1 to 3, we declare them pairwise distinct (e.g., $\neg(bitInd_1 = bitInd_2)$), and then we add axioms $\texttt{bva}(bitInd_3) \leftrightarrow \texttt{bvb}(bitInd_2)$ and $\texttt{bva}(bitInd_2) \leftrightarrow \texttt{bvb}(bitInd_1)$. Optionally, if, say, predicate $\texttt{less}_2$ has already been introduced in the instance, we might add axioms such as $\texttt{less}_2(bitInd_1)$, $\neg\texttt{less}_2(bitInd_2)$, and $\neg\texttt{less}_2(bitInd_3)$.

Although we use an abstraction of bit-vectors that assumes that they may have an infinite length, we do not lose soundness. Indeed, if we prove the validity of a formula by using such an interpretation of bit-vectors, it is also valid when the arguments to the bit-vectors range over fixed finite domains. However, we cannot claim completeness. For example, if a, b, c represent bit-vectors of length 1, the formula $a = b \vee a = c \vee b = c$ is valid, but it is not valid in our abstraction since its negation is satisfiable.

VI. Experimental results

We report here the experimental results of running the FOL theorem prover Vampire [22] for the two approaches, in four test cases. The first test case is our running example. The second test case is obtained from it by inserting a bug in the enabling logic. The third test case is similar to the first one, but the write operation happens via three ports, in three chunks. The fourth test case is obtained from the third by introducing a bug in the write data in the first chunk.

In tests 1 and 3, Vampire was able to prove equivalence using the algebraic approach in 5.7 and 4.4 seconds, respectively, whereas using the relational approach it was able to prove equivalence in 0.1 seconds in both tests. A digest of the statistics reported by Vampire is given in Figure 6. Note the dramatic improvement in Vampire's performance when using the relational approach: the number of inferences is several orders of magnitude less than for the algebraic translation. Furthermore, in tests 2 and 4 Vampire was able to detect the bug in 0 seconds for the relational approach, whereas for the algebraic one Vampire could not establish satisfiability in 1000 seconds.

We used Vampire on both translations in order to compare performance of the same theorem prover on these problems. We also ran iProver [14] which decides EPR formulas and was the best EPR prover in the last competition of first-order theorem provers on the relational translation. It was able to solve all problems in less than 0.01 second.

We have developed a prototype tool able to generate EPR instances using the relational approach from System Verilog descriptions of specification and implementation memory models. We have run two more real-life memory equivalence checking instances in the Darwin theorem prover [3]. Darwin was able to solve the instances in 11 seconds and 43 seconds respectively, and returned counter-examples (models) that we could use to debug the mismatches.

VII. Conclusions and future work

We have developed a modular, scalable, and efficient method for equivalence checking of real-life RTL and schematic descriptions of memories. We have introduced an axiomatization of address decoders that together with advanced circuit extraction capabilities drastically simplifies equivalence checking of real-life memories. We have introduced and compared two approaches of bit-selection in FOL, enabling the solving of equivalence problem instances where first-order axiomatization of memories needs to be combined with bit-vector reasoning.

Experimental results, produced using the FOL provers Vampire [22], iProver [14] and Darwin [3] on representative instances defined based on the analysis of real-life memory units in a recent chip design project at Intel, show that the equivalence of real-life memories can be verified within a few seconds.

Our approach makes possible modular verification and debugging by separating the verification of decoders from the rest of the verification. Mismatches caused by an incorrect decoder in the RTL or SCH models are detected when checking the decoding correctness property; mismatches caused by inconsistent read and write decoding are detected when checking the decoding consistency property in the SCH model; mismatches caused by differences in the read or write delays are detected when checking the memory correspondence invariant.

Although for the equivalence checking of memories we do not see a critical need for more efficient decision procedures, we plan to extend our axiomatization of bit-vectors in FOL to cover a larger set of bit-vector operations, and to develop decision procedures that combine our axiomatization with other powerful memory abstraction techniques.

We hope that our method can be extended and be useful for verifying complex protocols and high-level equivalence checking problems involving embedded memories.

An interesting future work is to modify the algebraic approach translation to make it suitable for SMT solvers able to work both with quantifiers and bit-vectors.

If a problem obtained by the relational approach is satisfiable, it is important to give an example showing that the design is incorrect. SAT solvers and first-order model builders, including iProver and Darwin, can be used to find finite models of satisfiable EPR problems. However, such models are of limited use for hardware design unless they are converted to concrete models based on bit-vectors and memories where all design constraints (such as bit-vector sizes) are satisfied. Translating such abstract first-order finite models into concrete models is an interesting and important future work.

References

[1] Abu-Haimed, H., D.L. Dill, S. Berezin. A refinement method for validity checking of quantified first-order formulas in hardware verification, FMCAD 2006.

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations, ICCD 2006.

[3] P. Baumgartner, A. Fuchs, H. de Nivelle, C. Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic, Journal of Applied Logic, 2007.

[4] Berkeley Logic Synthesis and Verification Group. ABC: a system for sequential synthesis and verification. Release 70930. http://www-cad.eecs.berkeley.edu/ alanmi/abc.

[5] Bjesse, P. Word-level sequential memory abstraction for model checking, FMCAD 2008.

[6] Bradley, A.R., Manna Z., Sipma H.B. What's decidable about arrays? VMCAI 2006.

[7] Bryant, R.E. Extraction of gate level models from transistor circuits by four valued symbolic analysis, CAD 1991.

[8] Clarke, E.M., O. Grumberg, D.A. Peled. *Model Checking*, MIT Press, 1999.

[9] Gannai, M.K., Gupta A., Ashar P. Verification of embedded memory systems using efficient memory modelling, DATE 2005.

[10] Huang, S.-Y., K.-T., Cheng. *Formal Equivalence Checking and Design Debugging*, Kluwer, 1998.

[11] Kaiss, D., S. Goldenberg, Z. Hanna, Z. Khasidashvili. Seqver: a sequential squivalence serifier for hardware designs, ICCD 2006.

[12] Khasidashvili, Z., D. Kaiss, D. Bustan. A compositional theory for post-reboot observational equivalence checking of hardware, FMCAD 2009.

[13] Koelbl, A., Burch J.R., Pixley C. Memory modeling in ESL-RTL equivalence checking, DAC 2007.

[14] Korovin, K. iProver–an instantiation-based theorem prover for first-order logic (system description), IJCAR, 2008.

[15] Korovin, K., Voronkov A. Integrating linear arithmetic into superposition calculus, CSL 2007.

[16] Kroening D., Strichman O. *Decision Procedures*, Springer EATCS, 2008.

[17] Manolios P., S.K. Srinivasan, D. Vroon. Automatic memory reductions for RTL model verification, ICCAD 2006.

[18] McCarthy, J., J. Painter. Correctness of a compiler for arithmetic expressions. Symposium in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society, 1967.

[19] de Moura, L.M., N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets, IJCAR 2008.

[20] Navarro-Perez, J.A., Voronkov A. Proof systems for effectively propositional logic, IJCAR 2008.

[21] Novakovsky, A., S. Shyman, Z. Hanna. High capacity and automatic functional extraction tool for industrial VLSI circuit designs, ICCAD 2002.

[22] Riazanov, A., A. Voronkov. The design and implementation of Vampire, AI Communications, 15(2-3):91–110, 2002.

A Compositional Theory for Post-Reboot Observational Equivalence Checking of Hardware

Zurab Khasidashvili, Daher Kaiss, Doron Bustan
Formal Technology and Logic Group
Intel, Israel Design Center, Haifa
{zurabk,dkaiss,dbustan}@iil.intel.com

***Abstract*— We propose an equivalence checking theory in a wider-than-usual sense. The theory shows how to combine Formal Equivalence Checking (FEC) of specification and implementation models with Assertion Based Verification (ABV) of the specification model, and with Reboot Sequence Checking (RSC) on both models, to ensure that the implementation model has the intended logic functionality. Here, FEC is performed to ensure that the input-output behavior of the models coincides in post-reboot states. ABV ensures that the specification model has the intended logic functionality captured by temporal assertions. RSC ensures deterministic behavior of the models after reboot. We propose a flexible compositional theory for FEC, an abstraction method for ABV, and a scalable algorithm for RSC, enabling performance of all three activities in a modular, compositional manner, and largely independently: FEC and ABV are performed without knowing the actual reboot sequence (and the respective initial states) of the two models; and FEC, ABV and RSC have the same observables.**

I. Introduction

In this work we are concerned with formal verification of the logic functionality of hardware designs. This task consists of several sub-tasks. A Register Transfer Level (RTL, e.g., SystemVerilog) description of a design is considered a golden model for the design. To ensure that it correctly implements the intended functionality, including logic, power, and security features, the RTL description, denoted by M_s (the specification model), is verified against temporal logic *assertions* that formally capture the design intent. This task is often called *Assertion Based Verification*, or ABV. The implementation model, denoted by M_i, is then generated based on the golden model either manually or by an automatic synthesis tool. The golden model M_s is compared against an implementation model M_i of the design. To ensure that the implementation model has the intended logic functionality, it must be proved "equivalent" to the RTL model. This task is called *Formal Equivalence Checking* [5], or FEC, of M_s and M_i. A useful concept of equivalence must preserve both the input-output behavior and the validity of the temporal assertions of M_s in M_i. Both FEC and ABV have their roots in and rely on *Model Checking* theory and algorithms [4].

In standard model checking, a model description includes a description of a set of initial states. This is essential for reasoning with both software and hardware systems. When modeling a hardware design, defining a set of initial states is not always trivial. This is because the powerup state for hardware cannot be determined uniquely. Once the design is powered up, a reboot sequence is injected into it through the design's inputs. This sequence brings the design into a set of states from which it should work correctly. As observed in [14], after a reboot sequence is applied on a design, the design is expected to behave deterministically. That is, applying a reboot sequence first and then some other input sequence, will always result in the same input-output behavior, no matter what the power-up state was before applying the reboot sequence. A reboot sequence that satisfies this condition is called *weakly synchronizing* [16]. Then, [14] shows that it is enough to prove that *there exists* a state in M_s and a state in M_i from which the input-output behavior of the two designs is the same, in order to conclude that the input-output behaviors of M_s and M_i are the same after any weakly synchronizing reboot sequence.

The RTL development flow works in a bottom-up fashion. That is, first the RTL is written for sub-units, then the units are integrated, followed by the clusters, and, only in the end, the whole system is built. In this process, it is often the case that the reboot sequence is created very late in the process, after all the units are integrated. However, RTL verification should be done in parallel with its development; it cannot wait until the reboot sequence of the entire system is ready. Therefore, there is a need to perform FEC and ABV before the entire system is built, and its reboot sequence is provided by the designer.

To solve the problem of the unavailability of the reboot sequence until the late stages of chip development, we have developed a theory which enables performing ABV and FEC without knowing the reboot sequence of the entire designs. We only need to assume that the designs belong to a well-defined class of observably X-initializable designs defined using 3-valued logic. This class contains the class of X-initializable designs widely used for hardware designs [5], [1]. In addition, we provide an algorithm for checking whether the reboot sequence is indeed observably X-initializing for the design. This activity is critical to proving that the reboot sequence is correct. This check should be performed on the actual reboot sequence of the design once it has been provided by the designer. We call this sequence a *global* reboot sequence. The ABV and FEC activities might be already completed by then, using *local* reboot sequences for the slices of the designs on which FEC and ABV operate. These sequences can be provided either by the designer or built automatically using an initialization algorithm, (e.g., [7]).

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

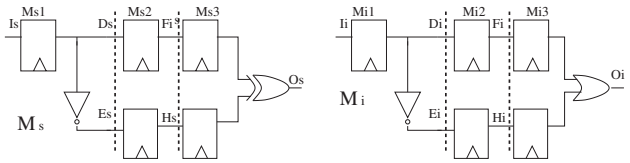


Fig. 1. Decomposition for compositional FEC.

To overcome the complexity limits of the formal engines, a compositional approach to FEC was developed in [9] where the models M_s and M_i are decomposed into pairs of corresponding slices. From proving equivalence of corresponding slices in M_s and M_i one can infer the equivalence of M_s and M_i. In [9], and unlike [5], [13], using properties over the inputs of a slice pair is allowed when proving their equivalence. The properties must have the form $\mathbf{G}\mathcal{B}$, where $\mathbf{G}$ is the linear temporal operator called *globally*, and $\mathcal{B}$ is a Boolean formula; we call such properties *combinational*. These properties will be referred to as *assumptions* when used during verification as constraints, and will be referred to as *assertions* when verified. The use of assumptions in compositional FEC enables keeping the slices small, within the complexity limits of the model checking engines. Figure 1 demonstrates this compositional approach. The compared models are decomposed at the signals $D_{\{s,i\}}$, $E_{\{s,i\}}$, $F_{\{s,i\}}$ and $H_{\{s,i\}}$ to form three slice pairs $(M_{s\{1,2,3\}}, M_{i\{1,2,3\}})$, separated by the dashed lines. To check the equivalence of (O_s, O_i) on slices M_{s3} and M_{i3}, an assumption of the form $\mathbf{G}(F_s \neq H_s)$ is added to the M_s model and used during the equivalence checking.

The main drawback of the previous theory [9] in the context of using assumptions is that it forced the assumptions to be verified on the partition used for FEC. That is, for every FEC slice with an assumption on its inputs, the assumption had to be proven on the outputs of a neighboring slice as an assertion. This approach limits the freedom of partitioning a system into modules, and in the context of equivalence checking it introduces a further dependency between the FEC and ABV activities, forcing them to work on the same slices of M_s. In the example of figure 1, the previous theory required verifying the assertion $\mathbf{G}(F_s \neq H_s)$ during ABV on the slice M_{s2} in the M_s model. In this case, the assertion cannot be verified as the slice needs to be expanded more and include the slice M_{s1}. Our theory lifts the limitation of the previous theory [9] by:

- allowing all linear temporal properties of the form $\mathbf{G}\psi$ to be used as assumptions on the cone boundaries;
- enabling assertions to be proved locally, using boundary assumptions, on slices that are different from slices used in FEC, and inferring their validity in the post-reboot states of the designs M_s and M_i;
- showing that the validity of these assumptions in the post-reboot states of M_s is enough for inferring the equivalence of M_s and M_i, provided all corresponding slice pairs are equivalent under the assumptions.

We propose an abstraction technique for proving assertions locally, and inferring their validity globally, on the post-reboot states of the design, without using the global reboot sequence. This approach is applicable to *proving* assertions of the form $\mathbf{G}\xi$ where ξ may be any linear temporal assertion [4]. We demonstrate that proving temporal assertions using abstraction techniques as presented in [3] is not sound in the presence of assumptions, if the design can be chosen arbitrarily from weakly synchronizable designs and the initial states are not known. We show that it becomes sound when the designs are limited to observably X-initializable designs.

The equivalence of M_s and M_i (with respect to the outputs) does not guarantee the preservation of the validity of the assertions of M_s in M_i [10]. To deal with this, the we assume that slice boundaries in M_s and M_i define the *observable* points. While in our theory ABV, FEC and RSC can be performed largely independently, one needs to *share the observables* in these activities. That is, we require the variables in assertions $\mathbf{G}\psi$ on M_s to be observables in M_s, and the reboot sequences of M_s and M_i to be observably X-initializing with respect to the the set of observables in M_s and M_i.

The paper is organized as follows: Section II includes some preliminaries. Section III presents motivating examples for a flexible compositional framework for performing FEC, ABV and RSC. Section IV presents novel notions of observable X-initialization, while section V presents an improved compositional framework for observably X-initializable designs, with a late reboot sequence. Experimental results are reported in Section VI. Section VII concludes the paper.

II. Preliminaries

A. Finite State Machines

An FSM [11] is a tuple $M = (S, I, O, \delta, \lambda)$ where S is a finite set of state variables, 2^S is a set of states, I is a finite set of input variables, 2^I is a set of inputs, O is a finite set of output variables, 2^O is the set of outputs, $\delta : 2^S \times 2^I \rightarrow 2^S$ is a transition function, and $\lambda : 2^S \rightarrow 2^O$, is an output function.

FSMs model electronic circuits in a trivial way, where S corresponds to flip-flops, I corresponds to inputs, O corresponds to outputs, δ corresponds to the combinational logic that drives the flip-flops, and λ corresponds to the combinational logic that drives the outputs.

Notation: Below w, u denote finite and infinite sequences of inputs. w^k is the $k+1'th$ element in w for $k \geq 0$. $|w|$ denotes the length w. For a set $\mathcal{S}$, $\mathcal{S}^*$ is the set of all finite sequences of elements in $\mathcal{S}$. $\pi = s^0 \xrightarrow{w^0} s^1 \xrightarrow{w^1} \ldots \xrightarrow{w^n} s^{n+1}$ is a *path of* M over an input sequence $w = w^0, w^1, \ldots, w^n$ if for every $0 \leq j \leq n$, $s^{j+1} = \delta(s^j, w^j)$. The paper deals with deterministic FSMs, thus, for every state s and input sequence w there exists a unique path of M on w that begins at s, denoted $path(s, w)$. $\delta(s, w)$ denotes the last state in $path(s, w)$, i.e. the state reached from s over w. $\lambda(\pi)$ denotes the output sequence $\lambda(s^0), \lambda(s^1), \ldots$.

State (s_s, s_i) of the product FSM $M_s \times M_i$ is an *equivalent state* (denoted by $s_s \cong s_i$) if for any input sequence w, $\lambda_s(path(s_s, w)) = \lambda_i(path(s_i, w))$. States s_s and s_i are then called equivalent states of M_s and M_i. State equivalence can

be generalized for two states s and t in the same model M such that $s \cong t$ iff they are equivalent in $M \times M$. A *weakly synchronizing* sequence [16] (ws-sequence for short) of an FSM M is an input sequence w that brings M from every state to a subset of equivalent states, called ws-states of M computed by w. A *reset* or *synchronizing* sequence w_r brings M from every state to a unique state s_r, called a *reset* or *synchronizing* state.

The post-reboot input-output behavior of hardware which is not weakly synchronizable is not deterministic and depends on the power-up state. Therefore, we expect all FSMs to be weakly synchronizable.

B. Observability

It is often the case that the specified behavior of an FSM cannot be observed on O. Therefore we define the observable set of variables O' that includes all the variables needed to define a valid behavior. Furthermore, when two FSMs are composed, the observables of the composition is the union of the observable sets, thus observables are not hidden in composition. For the rest of this paper O is used to denote the set of observable variables in the FSMs and λ maps states into subsets of the observables. The definition of $\lambda(\pi)$ for a path π is adjusted accordingly.

Since state equivalence w.r.t. the designs outputs does not preserve all of the specified behaviors, we define state equivalency w.r.t. the observables. That is, s_s and s_i are *observably equivalent*, written $s_s \cong s_i$, if for any input sequence w, $\lambda_s(path(s_s, w)) = \lambda_i(path(s_i, w))$. Then, an *observably synchronizing* sequence (or an *os-sequence*) of an FSM M is an input sequence w such that the set of states computed on w is observably equivalent. These states are called *os-states*.

C. Composition of FSMs

Notation: Let S be a set of variables and $S' \subseteq S$ a subset of S. Let f be an assignment to the variables of S. Then, $f|_{S'}$ is a restriction of f to the variables of S' s.t. for every $e \in S'$, $f(e) = f|_{S'}(e)$. Let S_1 and S_2 be disjoint sets of variables and let f_1 and f_2 be assignments to the variables of S_1 and S_2 respectively. We denote by $f_1 \cdot f_2$ the assignment to the variables of $S_1 \cup S_2$ s.t. $(f_1 \cdot f_2)|_{S_1} = f_1$ and $(f_1 \cdot f_2)|_{S_2} = f_2$.

Given FSMs $M_1 = (S_1, I_1, O_1, \delta_1, \lambda_1)$ and $M_2 = (S_2, I_2, O_2, \delta_2, \lambda_2)$ such that $S_1 \cap S_2 = \emptyset$ and $O_1 \cap O_2 = \emptyset$, their composition, denoted $M_1 \circ M_2$, is $M_c = (S_1 \cup S_2, I, O_1 \cup O_2, \delta_c, \lambda_1 \cdot \lambda_2)$, where $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ and δ_c is defined as follows: let $s_1 \in 2^{S_1}$, $s_2 \in 2^{S_2}$, and $i \in 2^I$; then, $\delta_c((s_1 \cdot s_2), i) = \delta_1(s_1, i|_{I_1} \cdot \lambda_2(s_2)|_{I_1}) \cdot \delta_2(s_2, i|_{I_2} \cdot \lambda_1(s_1)|_{I_2})$.

A transition $\delta_c((s_1 \cdot s_2), i)$ is over an assignment i to the input variables of $(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$. Since the FSMs are deterministic, a transition in M_c corresponds to unique transitions in M_1 and M_2 as follows: The transition corresponding to $\delta_c(s_1 \cdot s_2, i)$ in M_1 is $\delta_1(s_1, i|_{I_1} \cdot \lambda_2(s_2)|_{I_1})$ and in M_2 it is $\delta_2(s_2, i|_{I_2} \cdot \lambda_1(s_1)|_{I_2})$.

For compositional verification, consider a decomposition of $M_s = (S_s, I_s, O_s, \delta_s, \lambda_s)$ and $M_i = (S_i, I_i, O_i, \delta_i, \lambda_i)$ as two sets of sub-FSMs $\{M_s^j\}_j$ and $\{M_i^j\}_j$, where for every $M_s^j = (S_s^j, I_s^j, O_s^j, \delta_s^j, \lambda_s^j)$ there exists $M_i^j = (S_i^j, I_i^j, O_i^j, \delta_i^j, \lambda_i^j)$ that corresponds to it. The product $M_s \times M_i$ is a special case of composition where $I_s = I_i$, $O_s \cap I_i = \emptyset$ and $I_s \cap O_s = \emptyset$. Thus the composition is parallel, and $\delta_c((s_s \cdot s_i), i) = \delta_s(s_s, i) \cdot \delta_i(s_i, i)$. For this special case, we use (s_s, s_i) to denote a state of the product. We use $M_s^j \times M_i^j$ to denote corresponding sub-FSMs. A corresponding pair $M_s^j \times M_i^j$ shares the same interface, that is, there is a complete 1-1 onto mapping from the inputs M_s^j to the inputs of M_i^j and a complete 1-1 onto mapping from the observables of M_s^j to the observables of M_i^j.

D. Properties

This paper considers linear time properties. For the discussion in this paper, there is no need for a particular specification language. It is assumed that the relation $\pi \models \varphi$ (π satisfies φ) is defined for any path π and property φ. The satisfaction relations $s \models \varphi$ for a state s of an FSM M and property φ, is defined such that $s \models \varphi$ iff for every path π starting at s, $\pi \models \varphi$. In the presence of an assumption ψ over the inputs of M, we define $\langle s, \psi \rangle \models \varphi$ iff for every input sequence w that satisfies ψ, $\mathsf{path}(s, w) \models \varphi$. The only construct that is assumed to be in the specification language is $\mathbf{G}$, where a path π satisfies $\mathbf{G}\varphi$ iff every suffix of π satisfies φ. We assume that all of the variables in all properties are observables.

This paper considers properties written in M_s. We distinguish between two types of properties: Properties used as *assumptions* on the boundary of slices either in FEC or ABV, and *assertions* that specify a desired behavior of M_s. In this paper, assumptions and assertions are restricted to the form $\mathbf{G}\varphi$ where φ may be any linear time property. Note that assumptions may be used as assertions as well, and that all properties, assumptions and assertions, need to be proved. The limitations over the assumptions and assertions guarantees that the validity of these properties is closed under the transition function. In practice, most hardware specifications are written in $\mathbf{G}\varphi$ form; some specification languages, such as SVA [17], have an implicit $\mathbf{G}$ by default.

III. Post-reboot Observational Equivalence Theorem

The behaviors of the FSMs are observed in their post-reboot states. Therefore, FEC and ABV are performed on a pair (M, w) where M is an FSM and w is a reboot sequence. Previous work on ws-design focused only on FEC [14], and did not describe methods for ABV. As shown in this paper, with some adjustment, the results of this previous work can be adapted for ABV for properties of the form $\mathbf{G}\psi$.

The correct behavior of (M_s, w) and (M_i, w) and their post-reboot observational equivalence are defined as follows:

1) The *states computed on* w at M_s and M_i, $\{t \mid \exists r \in 2^{S_s} : t = \delta_s(r, w)\} \cup \{t \mid \exists r \in 2^{S_i} : t = \delta_i(r, w)\}$, are observably equivalent.
2) All assertions are satisfied by all *post-w states* $\{t \mid \exists r \in 2^{S_s}, \exists u \in I_s^* : t = \delta_s(r, w \cdot u)\}$ of M_s.

Theorem 3.1 gives sufficient conditions for performing FEC and ABV before the reboot sequence is available. The only task that is not performed before the reboot sequence is available is verifying that M_s and M_i are indeed observably synchronizable.

Theorem 3.1: Let $M_s \times M_i$ be FSMs. Further:

i Let w_s and w_i be observably synchronizing input sequences for M_s and M_i, respectively. Further, let $w = w_s \cdot w_i$, let $T_s = \{\delta_s(r, w) \,|\, r \in 2^{S_s}\}$, and $T_i = \{\delta_i(r, w) \,|\, r \in 2^{S_i}\}$.

ii Let M_s and M_i have an observably equivalent state pair, (s_s, s_i).

iii Let $\mathbf{G}\varphi$ be an assertion satisfied by a state r_s in M_s.

Then

1) The states in $T_s \cup T_i$ are observably equivalent.
2) $\mathbf{G}\varphi$ is satisfied by all post-w states of M_s and M_i.

Proof: The proof follows a similar one presented in [14], with some adjustments for ABV. By (i), w is observably synchronizing for both M_s and M_i. Let $t_s = \delta_s(s_s, w)$ and let $t_i = \delta_i(s_i, w)$, then $t_s \in T_s$ and $t_i \in T_i$. Then, (ii) implies that (t_s, t_i) is an observably equivalent state. Since w is observably synchronizing for both M_s and M_i, all states in T_s are observably equivalent and all states in T_i are observably equivalent. Therefore, all states in $T_s \cup T_i$ are observably equivalent. Let $e_s = \delta_s(r_s, w)$. Then, e_s is a state in T_s that satisfies $\mathbf{G}\varphi$. Since all states in T_s are observably equivalent, they satisfy $\mathbf{G}\varphi$ and thus every post-w state in M_s satisfies $\mathbf{G}\varphi$. Since the states in $T_s \cup T_i$ are observably equivalent, all states of T_i satisfy $\mathbf{G}\varphi$ and so do the post-w states of M_i. ■

For the above theorem to be useful in practice, for proving the equivalence of real-life hardware designs, we need:

A. A feasible abstraction technique that allows proving the validity of $\mathbf{G}\varphi$ in a state in M_s by proving its validity on a smaller sub-FSM of M_s.
B. A compositional framework that determines whether there is a pair of equivalent states in M_s and M_i.
C. A feasible, sufficient condition for deciding, in a compositional manner, whether an input sequence w is observably synchronizing for FSMs M_s and M_i.

A. Abstraction for ABV: motivation for improvements

For ABV, a compositional framework shall allow using abstraction techniques such as localization abstraction [3] to prove properties. Localization abstraction is a common way to abstract a design M in a proof of a property φ on a state s of M. It works as follows: define a sub-FSM (slice) M_1 of M, prove φ on state s_1 of M_1 *induced by* s and conclude its validity on state s of M. The soundness of this technique is implied from Lemma 3.2.

Lemma 3.2: [3] Let M_1 and M_2 be FSMs, and φ a property on variables in M_1. If a state s_1 of M_1 satisfies φ, then for every state s_2 of M_2, the state $s_1 \cdot s_2$ of $M_1 \circ M_2$ satisfies φ.

To enable small abstraction slices, such abstraction techniques often benefit from using properties already proved on the design (or on a sub-module) as assumptions in proofs of other properties. In the classical work on using assumptions in modular verification (e.g. [15]), one assumes that the initial states for which a property is proved on a sub-module M_1 are the states induced from the initial states of the entire system M. Since in our theory we want to perform ABV and FEC without knowing the reboot sequence (or the respective initial states) of the entire design M, we cannot use classical abstraction techniques, and we have to define validity of a property on a sub-FSM differently, where we replace *induced states* with *some states*. Let C be a set of properties on the inputs of M_1. We define:

- An assertion $\mathbf{G}\varphi$ is valid in M_1 constrained by C iff there *is a state* s_1 in M_1 such that $\langle s_1, \bigwedge_{\psi \in C} \psi \rangle \models \mathbf{G}\varphi$.

When C is empty, one can observe the following: If $\mathbf{G}\varphi$ is valid on M_1, then, by localization abstraction, there exists a state s in M that satisfies $\mathbf{G}\varphi$, and by Theorem 3.1, all os-states of M satisfy $\mathbf{G}\varphi$. Thus, localization abstraction can be used for ABV on observably synchronizable designs when no assumptions are used.

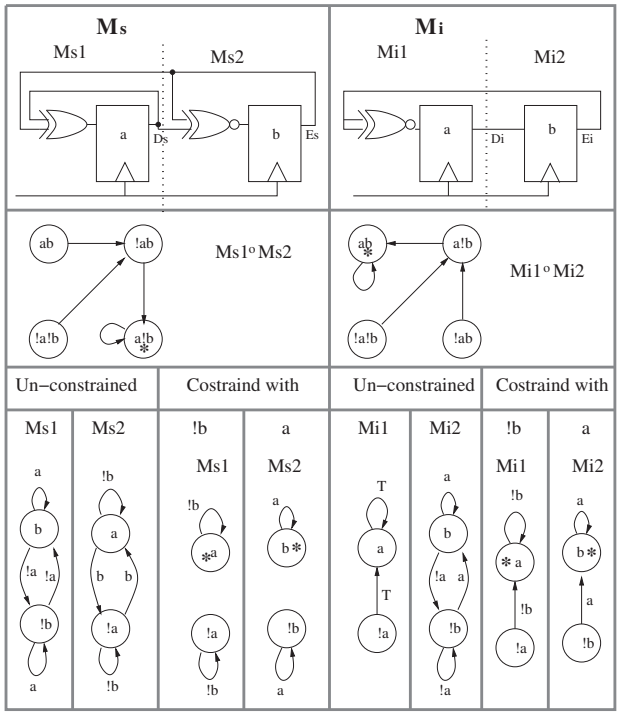


Fig. 2. Generalization of the 'induced-state' assume-guarantee principle [15] to 'some-state' assume-guarantee principle is not sound.

Using assumptions in compositional ABV is not sound in general for observably *synchronizable* designs. For example, consider M_s in Figure 2, where the observables are a and b. The property $\mathbf{G}a$ is valid in $Ms1 \circ Ms2$ (see state marked with $*$). Using $\mathbf{G}a$ as an assumption, enables the proof of $\mathbf{G}b$ on $Ms2$. This is because the state marked with $*$ in constrained $Ms2$ satisfies $\mathbf{G}b$. It is then concluded that $\mathbf{G}b$ is valid in M_s, which is false.

Note that the false property $\mathbf{G}b$ cannot be proven using Lemma 3.2 since $\mathbf{G}b$ is not valid on $Ms2$. Furthermore, $\mathbf{G}b$

cannot be proven using the "induced-state" assume-guarantee approach in [15] since the state marked with $*$ in $Ms2$ constrained by $\mathbf{G}b$ – the state on which $\mathbf{G}b$ is valid – is not the state induced by the state marked with $*$ in $Ms1 \circ Ms2$. Below we will show that the "some-state" assume-guarantee reasoning becomes sound when we restrict it to observably X*-initializable* designs.

B. Compositional FEC: motivation for improvements

A compositional framework with assumptions for FEC shall enable the following. Given a decomposition $\{M_s^j \times M_i^j\}_j$ of $M_s \times M_i$, let C^j be a set of assumptions on the inputs I_s^j of M_s^j. We would like to be able to say that if every pair of slices $M_s^j \times M_i^j$ is equivalent under C^j, then there exists an equivalent state in $M_s \times M_i$ as required in Theorem 3.1.

- A pair of slices $M_s^j \times M_i^j$ is equivalent under C^j, if there exists a state (s_s^j, s_i^j) of $M_s^j \times M_i^j$ such that for every input sequence w_j that satisfies the assumptions of C_j, $\lambda_s(path(s_s^j, w_j)) = \lambda_i(path(s_i^j, w_j))$.

In [9] a compositional approach for FEC was presented. However, it has the following limitation: For every assumption ψ^j in C^j, ψ^j is combinational, and there exists a neighboring pair of slices $M_s^k \times M_i^k$ on which ψ^j can be proved.

To understand this limitation, consider the example in Figure 3. Observe that both Ms and Mi are X-initializable: by keeping the inputs Is and Ii to 0 for four cycles, binary values will propagate to all latches in Ms and Mi, resulting in a full X-initialization of the designs. Suppose that for proving $\mathbf{G}(Os1 == Oi1)$ and $\mathbf{G}(Os2 == Oi2)$, the design is decomposed into three pairs of slices: $Ms1 \times Mi1$, $Ms2 \times Mi2$, and $Ms3 \times Mi3$. Then, $\mathbf{G}(Ds == Di)$ and $\mathbf{G}(Hs == Hi)$ are proved locally on $Ms1 \times Mi1$, $\mathbf{G}(Es == Ei)$ and $\mathbf{G}(Os1 == Oi1)$ are proved on $Ms2 \times Mi2$, and $\mathbf{G}(Bs == Oi2)$ is proved on $Ms3 \times Mi3$. Now, to complete a compositional proof, we need to prove $\mathbf{G}(Fs == Fi)$ on $Ms2 \times Mi2$ and prove $\mathbf{G}(Os2 == Oi2)$ on $Ms3 \times Mi3$. For proving $\mathbf{G}(Os2 == Oi2)$ on $Ms3 \times Mi3$ one needs a combinational assumption $\mathbf{G}(Es \neq Fs)$, which can be proved on $Ms2$, and this is allowed in [9]. However, for proving $\mathbf{G}(Fs == Fi)$ on $Ms2 \times Mi2$ one needs a non-combinational assumption $\mathbf{G}((\mathbf{N}Ds) \neq Hs)$, which is not allowed in [9] (here $\mathbf{N}$ is the linear temporal operator *next* [4]). In contrast, with the new theory, compositional FEC can be accomplished with the original decomposition by proving $\mathbf{G}((\mathbf{N}Ds) \neq Hs)$ on any sub-FSM of Ms, e.g., on $Ms1$.

Lifting the limitations imposed by [9] is not sound for all observably synchronizable designs. To see this, consider the example shown in Figure 2. There, both $Ms1 \circ Ms2$ and $Mi1 \circ Mi2$ are observably synchronizable (in fact they are synchronizable). The property $\mathbf{G}(a\&\&!b)$ is satisfied by all os-states of $Ms1 \circ Ms2$ and the property $\mathbf{G}(a\&\&b)$ is satisfied by the os-states of $Mi1 \circ Mi2$. Thus, M_s and M_i are not observably equivalent, since they do not have a pair of observably equivalent states. Nevertheless, with compositional reasoning using assumptions, it is possible to prove they are observably equivalent: using the assumption $\mathbf{G}a$ enables a proof that $Ms2 \times Mi2$ are observably equivalent: there is a pair of observably equivalent states marked with $*$ in constrained $Ms2$ and $Mi2$. Similarly, using the assumption $\mathbf{G}!b$, enables a proof that $Ms1 \times Mi1$ are observably equivalent; the pair of observably equivalent states is marked with $*$.

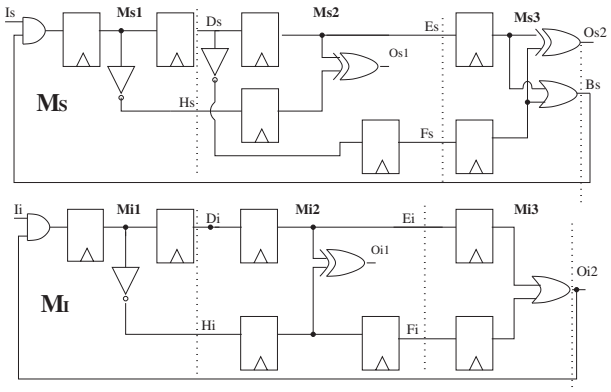


Fig. 3. Different properties with different cones in FEC and ABV.

C. Abstraction for RSC: motivation for improvements

There are no known formal, scalable methods for checking that a reboot sequence w is weakly or observably synchronizing. On the contrary, when the reboot sequence w is designed to be X-initializing (see the next section for the definition of X-initialization), checking that w is indeed X-initializing is feasible. For correct functionality of hardware a full X-initiallization is not required – for example, some memories need not be fully X-initialized if there is no read before write. Since full X-initialization requires extra initialization logic, most industrial designs are not X-initializable. To reason about such designs, below we formalize the concept of observable X-initialization and give an abstraction method for checking that a given reboot sequence is observably X-initializing. The later check is reduced to checking a number of local safety properties from known initial states.

The new method improves upon a previous formal method [10] for checking the correctness of a reboot sequence in that one now has full freedom in building a cone of influence for proving the local properties and using already proven properties (with variables in the cone boundary) as assumptions. Unlike [10], there is no need to restrict the cone to the sub-FSMs used in FEC decomposition in order to prove these local properties.

IV. Observable X-initialization

3-*valued simulation* [2] is a technique to compute an over approximation of the ws-states. In 3-valued modeling of FSMs, the variables' values can be one of the two *binary* values 0 or 1, or an *undefined* value X (elsewhere also denoted by $\perp$). Given a binary or ternary input sequence w, $\delta(s, w)$ now denotes the (ternary) state into which w brings M from state s; other FSM-related notation is similarly overloaded. The *unknown state* of M, denoted by χ, is the state in which

all the elements in S have the undefined value X. A *binary instance* of a ternary state s is a state obtained from s by assigning a binary value to all state variables that are X in s.

Boolean logic is extended to ternary logic as follows: $\neg X = X$, $X \vee 1 = 1 \vee X = 1$, $X \vee 0 = 0 \vee X = X$. These rules define a ternary extension to the transition function. The partial order $\leq_X$ over $\{0,1,X\} \times \{0,1,X\}$ includes the following pairs: $(0,0),(1,1),(X,X),(X,0),(X,1)$. This partial order is naturally extended to an order on ternary states: $s \leq_X t$ iff for every state variable $v \in S$, $s(v) \leq_X t(v)$. Similarly, since an input is an assignment to the input variables, it makes sense to write $i \leq_X j$, where i and j are ternary inputs of M. The order $\leq_X$ can now be extended to ternary input sequences and ternary paths of M.

The following lemma, stating the monotonicity of ternary simulation, is a basic tool in studying $\leq_X$:

Lemma 4.1: (X-Monotonicity, [1]) Let $s_1 = \delta(t_1, i_1)$ and $s_2 = \delta(t_2, i_2)$, where $t_1 \leq_X t_2$ and $i_1 \leq_X i_2$. Then $s_1 \leq_X s_2$.

An *X-initializing sequence [5]* of M is a sequence of inputs which, when applied to the unknown state of M, brings M into a binary state. Because of the monotonicity of 3-valued simulation, every X-initializing sequence of M is synchronizing, and thus is a ws-sequence of M.

Definition 4.2:

- A ternary state s of an FSM M is *deterministic* if for every binary input sequence w of M, $\lambda(\delta(s,w))$ is binary.
- An input sequence w of M is an *observably X-initializing* sequence if w brings M from the χ state into a deterministic state s. The ternary state s, as well as its binary instances, are *observably X-initial* states of M.

Lemma 4.3: All binary instances of a ternary deterministic state are observably equivalent.

Proof: Let s and t be binary instances of a ternary deterministic state e of an FSM M, let w be any input sequence of M, and let $s' = \delta(w,s)$, $t' = \delta(w,t)$, and $e' = \delta(w,e)$. By the X-Monotonicity Lemma, s' and t' are binary instances of ternary state e'. Since e is deterministic, the observables of M at ternary state e' are binary, and their values coincide with the values of the corresponding observables in s' and t'. Thus $s' \cong t'$. ∎

Hence an observably X-initializing sequence brings an FSM from the χ-state, and from any binary state, into a set of observably equivalent states, and thus is observably synchronizing. Since every binary state is deterministic, any X-initializing sequence is observably X-initializing.

Notation: For an FSM $M = M_1 \cdot M_2$ and a set $\mathcal{S}$ of states in M, the projection $\mathcal{S}|_{M_1}$ of $\mathcal{S}$ on M_1 is the set $\{s_1 | \exists s_2 \in M_2 : s_1 \cdot s_2 \in \mathcal{S}\}$. For a path $\pi = s_1^0 \cdot s_2^0 \xrightarrow{w^0} s_1^1 \cdot s_2^1 \xrightarrow{w^1} \ldots \xrightarrow{w^n} s_1^{n+1} \cdot s_2^{n+1}$ in M, the projection $\pi|_{M_1}$ of π on M_1 is the path $s_1^0 \xrightarrow{w_1^0} s_1^1 \ldots \xrightarrow{w_1^n} s_1^{n+1}$, where $w_1^j = w^j|_{I_1} \cdot \lambda_2(s_2^j)|_{I_1}$. For an input sequence w, a state $s_1^0 \cdot s_2^0$ of M, and $\mathsf{path}(s_1^0 \cdot s_2^0, w) = s_1^0 \cdot s_2^0 \xrightarrow{w^0} s_1^1 \cdot s_2^1 \xrightarrow{w^1} \ldots \xrightarrow{w^n} s_1^{n+1} \cdot s_2^{n+1}$ in M, the projection $(s_1^0 \cdot s_2^0, w)|_{I_1}$ of w on I_1 is the sequence $w^0|_{I_1} \cdot \lambda_2(s_2^0)|_{I_1}, \ldots, w^n|_{I_1} \cdot \lambda_2(s_2^n)|_{I_1}$.

Lemma 4.4: Let w be an observably X-initializing sequence of $M = M_1 \cdot M_2$. Let s be a state of M. Then, $\delta_1(\chi_1, (s,w)|_{I_1}) \geq_X \delta(\chi, w)|_{M_1}$, where, $\chi_1 = \chi|_{M_1}$.

Proof: By induction over the length $|w|$ of w: Let $w_1 = (s,w)|_{I_1}$. *Base:* For $|w| = 0$, $(s,w)|_{I_1} = \epsilon$, and $\delta_1(\chi_1, (s,w)|_{I_1}) = \chi_1 = \delta(\chi,w)|_{M_1}$. *Step:* For $|w| = n$, let $p_1 \cdot p_2 = \delta(s, w^{0..n-1})$, $q_1 = \delta_1(\chi_1, w_1^{0..n-1})$ and $t_1 \cdot t_2 = \delta(\chi, w^{0..n-1})$. By induction hypothesis, $q_1 \geq_X t_1$. By monotonicity, $p_1 \cdot p_2 \geq_X t_1 \cdot t_2$. Let $q_1' = \delta_1(q_1, w_1^n)$ and $t_1' \cdot t_2' = \delta(t_1 \cdot t_2, w^n)$. Since $w_1^n = w^n|_{I_1} \cdot \lambda_2(p_2)|_{I_1}$, $q_1' = \delta_1(q_1, w^n|_{I_1} \cdot \lambda_2(p_2)|_{I_1})$. By definition, $t_1' = \delta_1(t_1, w^n|_{I_1} \cdot \lambda_2(t_2)|_{I_1})$. Since $p_2 \geq_X t_2$ and $q_1 \geq_X t_1$, monotonicity implies, $q_1' \geq_X t_1'$. ∎

V. Compositional framework for observably X-initializable designs

In this section we present the main results of the paper.

Notation: Let w be an input sequence. Then $w^{k..j}$ denotes the sub-sequence of w that begins at w^k and ends at w^j, and $w^{k..}$ denotes the suffix of w that begins at w^k. Further, $w^{k..k-1}$ and ϵ denote the empty sequence.

A. Abstraction for ABV: a flexible solution

Theorem 5.3 shows that for observably X-initializable designs, it is sound to use an assumption $\mathbf{G}\psi$ in localization abstraction in the proof of a property $\mathbf{G}\varphi$, as long as $\mathbf{G}\psi$ monitors the inputs of slice M_1 on which $\mathbf{G}\varphi$ is proved.

Lemma 5.1: Let $M = M_1 \circ M_2$ be an observably X-initializable FSM. Let w be an observably X-initializing sequence for M. Let $s_1 \cdot s_2$ and $t_1 \cdot t_2$ be states of M. Let $w_1 = (s_1 \cdot s_2, w)|_{I_1}$. Then $\lambda_1(\delta_1(s_1, w_1)) = \lambda_1(\delta_1(t_1, w_1))$.

Proof: Since w is observably X-initializing, $\delta(\chi, w)$ is a deterministic state. By Lemma 4.3, for every pair of binary instances $p_1 \cdot p_2$ and $q_1 \cdot q_2$ of $\delta(\chi, w)$, $\lambda(p_1 \cdot p_2) = \lambda(q_1 \cdot q_2)$. In particular $\lambda_1(p_1) = \lambda_1(q_1)$. By Lemma 4.4, $\delta_1(\chi_1, w_1) \geq \delta(\chi, w)|_{M_1}$. Thus, for every pair of binary instances p_1 and q_1 of $\delta_1(\chi_1, w_1)$, $\lambda_1(p_1) = \lambda_1(q_1)$. Since $\delta(s_1, w_1)$ and $\delta_1(t_1, w_1)$ are binary instances of $\delta_1(\chi_1, w_1)$, $\lambda_1(\delta(s_1, w_1)) = \lambda_1(\delta_1(t_1, w_1))$. ∎

Since every extension of an observably X-initializing sequence is an observably X-initializing sequence, we can conclude Corollary 5.2.

Corollary 5.2: Let $M = M_1 \circ M_2$ be an observably X-initializable FSM. Let w be an observably X-initializing sequence for M. Let $s_1 \cdot s_2$ and $t_1 \cdot t_2$ be states of M. Let u be an input sequence and let $w_1 = (s_1 \cdot s_2, w \cdot u)|_{I_1}$. Then for every $j \geq |w|$, $\lambda_1(\delta_1(s_1, w_1^j)) = \lambda_1(\delta_1(t_1, w_1^j))$.

Theorem 3.1 implies that for observably X-initializable design M and a property $\mathbf{G}\varphi$, if $M \not\models \mathbf{G}\varphi$, then for every state s there exists an input sequence u such that $\mathsf{path}(s,u) \not\models \mathbf{G}\varphi$. Let w be an observably X-initializing sequence for M, then for every state s there exists an input sequence u such that $\mathsf{path}(s, w \cdot u) \not\models \mathbf{G}\varphi$, furthermore, $\mathsf{path}(s, w \cdot u)$ has a suffix $\mathsf{path}(s, w \cdot u)^{j..}$ such that $\mathsf{path}(s, w \cdot u)^{j..} \not\models \mathbf{G}\varphi$ and $j > |w|$.

Theorem 5.3: Let $M = M_1 \circ M_2$ be an observably X-initializable FSM. Let w be an observably X-initializing sequence for M. Let $\mathbf{G}\psi$ be a property whose variables are contained in I_1 and let $\mathbf{G}\varphi$ be a property whose variables are

contained in O_1. Let s be a state of M such that $s \models \mathbf{G}\psi$. Then, if $M \not\models \mathbf{G}\varphi$, then for every state t_1 of M_1, $\langle t_1, \mathbf{G}\psi\rangle \not\models \mathbf{G}\varphi$.

Proof: Let $w_1 = \mathsf{path}(s, w \cdot u)|_{I_1}$. Then $w_1 \models \mathbf{G}\psi$. Let $s_1 = s|_{M_1}$, then $\mathsf{path}(s_1, w_1) \not\models \mathbf{G}\varphi$, furthermore, for some $j > |w|$, $\mathsf{path}(s_1, w_1)^{j..} \not\models \mathbf{G}\varphi$. By Corollary 5.2, $\lambda_1(\mathsf{path}(t_1, w_1)^{j..}) = \lambda_1(\mathsf{path}(s_1, w_1)^{j..})$ thus $\mathsf{path}(t_1, w_1)^{j..} \not\models \mathbf{G}\varphi$. This implies that $\mathsf{path}(t_1, w_1) \not\models \mathbf{G}\varphi$. Since $w_1 \models \mathbf{G}\psi$, we conclude that $\langle t_1, \mathbf{G}\psi\rangle \not\models \mathbf{G}\varphi$. ■

B. Compositional FEC: a flexible solution

Theorem 5.4 below shows that for observably X-initializable FSMs, it is sound to prove an assumption of M_s separately from M_i, without restricting the slice it is proved on to a slice that is used for FEC. This assumption is used in proving that a pair of slices are equivalent; it is required that the assumption's variables are inputs of the slice of M_s.

Theorem 5.4: Let $\{M_s^j \times M_i^j\}_j$ be a decomposition of observably X-initializable FSMs M_s and M_i. Let C^j be a set of assumptions on the inputs of M_s^j, and let $\mathbf{G}\psi^j = \wedge C^j$. Further, assume:

(a) There is a state s_s in M_s that satisfies $\mathbf{G}\psi^j$.
(b) In every pair of slices $M_s^j \times M_i^j$ there is a state pair (t_s^j, t_i^j) such that $t_s^j \cong_{\psi^j} t_i^j$.

Then, M_s and M_i have observably equivalent states.

Proof: Let w be an observably X-initializing sequence of M_s and M_i, let u be an infinite input sequence of M_s and M_i, let $z_s = \delta_s(s_s, w)$, and let $w_j = (s_s, w)|_{I_s^j}$ and $u_j = (z_s, u)|_{I_s^j}$. Let $y_s^j = \delta_s^j(t_s^j, w_j)$ and $y_i^j = \delta_i^j(t_i^j, w_j)$. Then, for every M_s^j: Since $s_s \models \mathbf{G}\psi^j$, $w_j \cdot u_j \models \mathbf{G}\psi^j$. By monotonicity, $y_s^j \geq_X \delta_s^j(\chi_s^j, w_j)$. By Lemma 4.4, $y_s^j \geq_X \delta_s(\chi_s, w)|_{M_s^j}$. Since $y_s^j \geq_X \delta_s(\chi_s, w)|_{M_s^j}$ for all j, the concatenation y_s of all y_s^j satisfies $y_s \geq_X \delta_s(\chi_s, w)$. Since z_s and y_s are instances of $\delta_s(\chi_s, w)$ and w is observably X-initializing, $z_s \cong y_s$ by Lemma 4.3. Thus $(y_s, u)|_{I_s^j} = u_j$.

Let y_i be the concatenation of all y_i^j. We prove $y_s \cong y_i$ by showing $\lambda_s(\delta_s(y_s, u^{0..k-1})) = \lambda_i(\delta_i(y_i, u^{0..k-1}))$, by induction on k. *Base:* Since $t_s^j \cong_{\psi^j} t_i^j$ and y_s^j and y_i^j are states on paths $\mathsf{path}(t_s^j, w_j \cdot u_j)$ and $\mathsf{path}(t_i^j, w_j \cdot u_j)$ and are obtained by applying w_j to t_s^j and t_i^j, respectively, and since $w_j \cdot u_j \models \mathbf{G}\psi^j$, we get by definition of $\cong_\psi$ and by the construction of y_s and y_i from y_s^j and y_i^j that $\lambda_s(\delta_s(y_s, \epsilon)) = \lambda_i(\delta_i(y_i, \epsilon))$. *Step:* Assume that $\lambda_s(\delta_s(y_s, u^{0..k-1})) = \lambda_i(\delta_i(y_i, u^{0..k-1}))$. Suppose on the contrary that for some observable (o_s, o_i) of a pair of slices $M_s^{j_0} \times M_i^{j_0}$, $\lambda_s(\delta_s(y_s, u^{0..k}))|_{o_s} \neq \lambda_i(\delta_i(y_i, u^{0..k}))|_{o_i}$. Since there is at least one cycle delay between the inputs and the outputs of $M_s^{j_0}$ and $M_i^{j_0}$, the two sub-FSMs receive the same inputs at cycles $0, ..., k-1$ (and along w_{j_0}). Thus by $t_s^{j_0} \cong_{\psi^{j_0}} t_i^{j_0}$ and $w_{j_0} \cdot u_{j_0} \models \mathbf{G}\psi^{j_0}$, all outputs of $M_s^{j_0}$ and $M_i^{j_0}$ must have the same values in $\delta_s(y_s, u^{0..k})$ and $\delta_i(y_i, u^{0..k})$ - contradiction. ■

C. Abstraction for RSC: a flexible solution

Theorem 3.1 needs a feasible, sufficient condition for deciding, in a compositional manner, whether an input sequence w is observably synchronizing for FSMs M_s and M_i. Given a design M, a reboot sequence w and a set of observables O, verifying the reboot sequence is done in two steps:

1) Computing the ternary state $s_e = \delta(\chi, w)$ using ordinary 3-valued simulation. This is done on the whole system M. 3-valued simulation is computationally inexpensive; it is feasible to run it on the whole system.
2) Verifying that the end state s_e is deterministic, i.e., that no input sequence may cause X to appear on the signals of O. This is a more expensive task essentially equivalent to model-checking of an FSM encoded over $\{0, 1, X\}$ (via dual-rail modeling [2]). It can be done in a compositional manner, where an FSM is decomposed on observables. Then, in every sub-FSM, it is proven that starting from the projection of s_e, for any observable o of the slice, $\mathbf{G}(o \neq X)$.

Note that despite the need to work with dual-rail encoding (which "doubles" the number of propositional variables in the model-checking instance) there is no noticeable impact on the performance, since the X value is represented by constants in each rail, and constant propagation reduces the size of the instance significantly (see [8] for experimental evidence).

VI. Experimental results

To understand the usefulness of our theory in practice, in this section we explain the ABV methodology currently used in a large microprocessor project at Intel.

Contemporary multi-core microprocessors are built in a hierarchical manner. In every core, the highest hierarchy is a cluster (for example, floating-point arithmetic). Each cluster is divided into units and each unit into FUBs (Functional Blocks). FEC is performed at tha FUB level – meaning that the RTL and schematic models of each FUB are divided into sub-FSMs and compared (using boundary properties). On the other hand, ABV of FUB-level properties is performed at unit level. The reason for this is that it is often impossible to verify assertions at the FUB level because a property's required cone of influence stretches beyond the boundaries of the FUB.

ABV is performed in two stages. At the first stage, an automatic initialization algorithm [7] is applied to a given unit with a timeout. Often some of the latches remain un-unitialized, and specific values are set to them randomly or based on some knowledge of "legal initial states". Then the properties are model checked from this initial state, after building a relevant cone of influence. Properties for which a counterexample can be built on the primary inputs of the unit are debugged. If there are unresolved properties in the first stage (caused by a timeout or out-of-memory), then in the second stage a reboot sequence provided by the unit designer is applied to the unit from the X-state. Note that this reboot sequence is not induced from the real full-chip reboot sequence – the unit reboot sequence is built based on knowledge of the unit and is much shorter than the full-chip reboot sequence. Again, the X-values that normally remain in the unit after applying the unit reboot sequence are set to binary values, and the properties that were left unresolved in the first stage

FUBs	# all props	# used in FEC	ABV on slices valid/false/unr	ABV on units valid/false/unr
FUB1	18	17	0/1/17	3/15/0
FUB2	172	49	16/155/1	68/13/91
FUB3	26	9	7/16/3	26/0/0
FUB4	36	31	12/22/0	35/1/0
FUB5	3	2	1/2/0	3/0/0

Fig. 4. Comparing ABV results with methodologies based on an old [9] and the new theory.

are re-verified. According to our theory, defining the initial state as above is safe in that any property proven valid from some state of an abstracted model is guaranteed to be valid in post-reboot states of the full-chip. False negatives are possible because of missing assumptions or because of the poor quality of the initialization. From the project's experience, the quality of formal initialization is significantly higher (with respect to false negatives) than that of initialization that uses the reboot sequence provided by the designer. This is why formal initialization is chosen as the initialization method in the first stage of ABV. The project's experience shows that in 60% of the units, the second stage in ABV is not necessary.

Recall that the earlier compositional FEC theory [9] required proving the assertions used in FEC on the local slices (thus on sub-FSMs of the respective FUBs), and no sequential assumptions were allowed. Table 4 compares ABV results on 5 FUBs, performed using the old and new theories (on slices used in FEC and at unit level, respectively). The second column shows the number of assertions in the FUB, the third column shows how many of these assertions were used in FEC as assumptions, and the remaining two columns show the number of properties proven valid, falsifiable, or which remained unresolved, using the old and new methodologies. It can be seen that a large portion of the properties that couldn't be proven with the old methodology can be proven with the new methodology. The table reports work in progress – some of the properties are falsifiable (due to missing assumptions) or remain unresolved.

VII. Conclusions

We proposed a compositional theory for combining FEC, ABV and RSC, which ensures that the observable behavior of the specification model will be preserved in the implementation model after the reboot sequence. Further, for observably X-initializable designs introduced in this paper (this class contains a large class of industrial designs), we showed that, compared to previous work [9], assumptions can be used with fewer limitations in both FEC and ABV before a global reboot sequence is available. Finally, we provide an improved (cf. [10]) algorithm which allows verifying the reboot sequence locally on small slices of the designs.

Unlike the classical abstraction techniques in Model Checking [4], for observably X-initializable designs (with observable X-initial states as the initial states), we show that the initial states of an abstracted module do not need to be the states induced from the initial states of the full system for the abstraction to be sound. Thus, if the requirements of the full system from the initial states of the module change, or if the module is plugged into a different system, with our theory there is no longer any need to re-verify the module with the initial states induced from the new system.

In practice, in large chip-design projects at Intel, the full-chip reboot sequence is available only at a late stage of the design, close to tape-out, and by then FEC and ABV are completed, and in fact FEC and ABV regressions are optimized in that the best performing verification strategies and cone partitioning are used. Therefore, synchronizing the FEC and ABV with the initial states computed by X-initilization using the full-chip reboot sequence represents a big overhead for FEC and ABV. For example, using full-chip initial states may lead to false negatives in ABV when there are initialization-related bugs anywhere in the full chip or if the reboot sequence is not correct (or becomes incorrect after bug-fixes or late changes in the design). Supported by our theory, two recent large chip-design projects with non-state-matching RTL and SCH taped-out without ever using full-chip initial states in FEC or ABV.

References

[1] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations, ICCD 2006.

[2] R.E. Bryant. A methodology for hardware verification based on logic simulation, JACM 38, 1991.

[3] E.M. Clarke, O. Grumberg, D.E. Long. Model Checking and Modular Verification, ACM Trans. Program. Lang. Syst., 1994.

[4] E.M. Clarke, O. Grumberg, D.A. Peled. *Model Checking*, MIT Press, 1999.

[5] S.-Y. Huang, K.-T., Cheng. *Formal Equivalence Checking and Design Debugging*, Kluwer, 1998.

[6] D. Kaiss, S. Goldenberg, Z. Hanna, Z. Khasidashvili. Seqver: a sequential equivalence verifier for hardware designs, ICCD 2006.

[7] D. Kaiss, M. Skaba, Z. Hanna, Z. Khasidashvili. Industrial strength SAT-based alignability algorithm for hardware equivalence verification, FMCAD 2007.

[8] Z. Khasidashvili, Z. Hanna. SAT-based methods for sequential hardware equivalence verification without synchronization, BMC 2003.

[9] Z. Khasidashvili, M. Skaba, D. Kaiss, Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints, ICCAD 2004.

[10] Z. Khasidashvili, M. Skaba, D. Kaiss, Z Hanna. Post-reboot equivalence and compositional verification of hardware, FMCAD 2006.

[11] Z. Kohavi. *Switching and Finite Automata Theory*, McGraw-Hill, 1978.

[12] K. L. McMillan. Circular compositional reasoning about liveness, CHARME 1999.

[13] 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.

[14] C. Pixley. A theory and implementation of sequential hardware equivalence, IEEE transactions on CAD, 1992.

[15] A. Pnueli. In transition from global to modular temporal reasoning about programs, In: Logics and Models of Concurrent Systems, Springer LNCS, vol. F-13, NATO ASI, 1985.

[16] I. Pomerance, S. M. Reddy. On removing redundancies from synchronous sequential circuits with synchronizing sequences, IEEE Trans. Comput., 1996.

[17] IEEE Computer Society. 1800: IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language. IEEE 2005.

Scaling VLSI Design Debugging with Interpolation

Brian Keng[1], Andreas Veneris[1,2]
[1]Department of Electrical and Computer Engineering, University of Toronto
[2]Department of Computer Science, University of Toronto
{briank, veneris}@eecg.toronto.edu

***Abstract*—Given an erroneous design, functional verification returns an error trace exhibiting a mismatch between the specification and the implementation of a design. Automated design debugging uses these error traces to identify potentially erroneous modules causing the error. With the increasing size and complexity of modern VLSI designs, error traces have become longer and harder to analyze. At the same time, design debugging has become one of the most resource-intensive steps in the chip design cycle. This work proposes a scalable SAT-based design debugging algorithm that uses interpolants to over-approximate sets of constraints that model the erroneous behavior. The algorithm partitions the original problem into a sequence of smaller subproblems by using subsections of the error trace that are examined iteratively. This is made possible by using interpolants to properly constrain the erroneous behavior for each subproblem, significantly reducing the number of simultaneous time-frames examined in the error trace. The described method is shown to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants. Experiments on real designs show a 57% reduction in memory and 23% decrease in run-time compared to previous work.**

I. Introduction

The aim of functional verification is to determine whether the implementation of a design conforms to its specification. If the design is found to be buggy, an *error trace* is returned which exposes the erroneous behavior. Due to the increasing size and complexity of modern designs, error traces generated from functional verification tools [1]–[5] can be thousands of clock cycles long [6]. This places a large burden on the engineer to examine the error trace and identify potential design bugs causing the erroneous behavior.

The task of identifying these potential error suspects in the design is called *design debugging*. This process begins after verification fails and an error trace is produced. An engineer then has to analyze this error trace, typically through a waveform viewer or some other graphical representation, a process today that is predominantly manual. This task can consume months of the verification effort and as much as 30% of the total time to design a Very Large Scale Integration (VLSI) chip [7]. Due to this fact, automated debugging methodologies remain of great interest to both the research and industrial communities.

Many automated debugging techniques have been developed to aid the engineer in this task. Simulation-based techniques [8], [9] have been extensively studied in the past and can be effective in certain situations. More recently, formal frameworks [10]–[15], such as those based on Boolean Satisfiability (SAT), have achieved significant advancements in design debugging.

Given a sequential design, SAT-based debugging techniques require a time-frame expansion of the circuit. This involves replicating the combinational component of the circuit such that the next-state variables of time-frame i are connected to the current-state variables of time-frame $i+1$ for the length of the error trace. For large designs and long error traces, this approach produces SAT instances which may not be practically viable because of the large memory footprint. Reducing the requirements for these techniques without sacrificing performance becomes an urgent necessity towards the development of scalable automated debugging tools.

In this work, we propose a novel scalable SAT-based design debugging algorithm which leverages interpolants to over-approximate sets of constraints that model the erroneous behavior, significantly reducing the memory-intensive circuit replication at any given time. This is accomplished by dividing the error trace into several parts, or *windows*, and analyzing each window of time-frames separately. To allow for each window to be properly constrained with the erroneous behavior, interpolants are used to over-approximate sets of constraints that model time-frames within close proximity to the observed error. The analysis begins with a window at the end of the error trace. If the analysis does not yield complete results, it proceeds by moving the window backwards iteratively. The interpolant is calculated from the unsatisfiable (UNSAT) core resulting from previously analyzed windows. The net result of this iterative methodology is a significant reduction in memory requirements and improvements in run-time.

The described method is shown to find all error locations whose functions can be modified to correct the erroneous behavior for a given error trace and number of errors. Additionally, a technique to generate multiple interpolants is introduced to reduce the number of error locations returned, thus improving the quality of the debugging results.

An extensive set of experiments on large hardware designs and long error traces illustrates the benefits of this work. It is shown that a conservative partitioning of the error trace yields an average 34% reduction in memory and 24% reduction in run-time compared to traditional SAT-based debugging, while the number of returned error locations is only increased on average by 1% of the total number of suspects. For a more aggressive partitioning scheme, averages of 57% reduction in memory and 23% reduction in run-time are achieved at the cost of increasing the relative number of error locations returned by 2%. This favorable trade-off between resolution and performance allows for scaling of existing SAT-based debugging methodologies to handle modern VLSI designs.

The remaining sections of the paper are organized as follows. Section II defines notation as well as background on debugging, UNSAT cores and interpolants. Section III illustrates the use of interpolants in partitioning the debugging problem. Section IV presents experimental results and

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

Section V concludes this work.

II. Preliminaries

A. Notation and Design Debugging

This section provides notation used throughout this paper and background information on design debugging.

The letters x, y and s refer to the primary inputs, primary outputs and state elements of a sequential circuit. x^i, y^i and s^i denote Boolean vectors in the i^{th} clock-cycle, or *time-frame*, of a sequential operation of a circuit. Similarly, x^i_j, y^i_j and s^i_j refer to the j^{th} indexed bit in the i^{th} Boolean vector. Finally, X^i, Y^i and S^i denote a predicate for the i^{th} clock cycle.

The behavior of a sequential circuit C can be described formally by a transition relation, $T(s^i, s^{i+1}, x^i, y^i)$, which is `true` if and only if given the current-state s^i, applying primary inputs x^i to C will generate primary outputs y^i and the next-state s^{i+1}.

Design debugging aims to find all error locations, or *suspects*, which could potentially explain the erroneous behavior demonstrated in a given error trace [9]. In this work, we define a design debugging method to be *complete* for a given error trace and number of errors, if and only if it returns all suspects whose functions can be modified separately to fix the erroneous behavior in the error trace. The *resolution* of a debugging method refers to the total number of suspects returned, where fewer suspects correspond to better resolution.

Formally, we define $\mathcal{V}_0^k$ of length k+1, as an error trace for clock-cycles 0 to k to consist of an initial state predicate, a vector of primary input predicates and a vector of *correct* or *expected* primary output predicates from 0 to k, which can be written as follows:

$$\mathcal{V}_0^k = \langle S^0, \langle X^0, \ldots, X^k \rangle, \langle Y^0, \ldots, Y^k \rangle \rangle \tag{1}$$

A *window* of an error trace from clock-cycles p to q, is defined as a consecutive subsequence of an error trace, $\mathcal{V}_p^q = \langle S^p, \langle X^p, \ldots, X^q \rangle, \langle Y^p, \ldots, Y^q \rangle \rangle$, where S^p is calculated by applying the initial state predicate and the first p primary input predicates to the transition relation, i.e. simulating the erroneous circuit for p cycles. Using this notation, a *prefix window* of length p for this trace can be written as $\mathcal{V}_0^{p-1}$ and a *suffix window* of length $k-p+1$ can be written as $\mathcal{V}_p^k$. We will occasionally omit the term window and use the term suffix or prefix in place of suffix window or prefix window respectively.

For this work, we assume that the error is first observed in the last clock cycle of the error trace. If this is not the case, a shorter error trace can be trivially generated by taking the shortest prefix that exhibits the erroneous behavior.

B. SAT-based Design Debugging

This section briefly describes background and notation for SAT-based design debugging that is relevant to our contribution. SAT-based design debugging [10] is a complete method that encodes the design debugging problem into a SAT instance for a given error trace and number of errors. The satisfying assignments of the SAT instance correspond to suspects which can be replaced with non-deterministic functions to correct the erroneous behavior in the error trace. The SAT instance is created in several steps. First, the transition relation is enhanced by introducing a set of *suspect variables*, $E = \{e_0, \ldots, e_n\}$, where each e_i corresponds to the i^{th} potential error location (gate, module etc.). The suspect variables are then added to the transition relation such that if $e_i = 1$ then the i^{th} potential error location is disconnected from its fan-in and become free variables. This can be achieved either through a hardware construction using multiplexors, or directly in conjunctive normal form (CNF). Note that each e_i can correspond to multiple gates depending on the type of the error location. The enhanced transition relation is denoted by $T_{en}(s^i, s^{i+1}, x^i, y^i, E)$.

Next, T_{en} is unrolled as a time-frame expanded model for the length of the error trace, such that the next-state of time-frame i is connected to the current-state of time-frame $i+1$. Note that the suspect variables are not replicated since they represent the same location regardless of the time-frame. The error trace predicates are then applied to the initial state, input and output variables of the replicated enhanced transition relation.

Finally, the number of simultaneous active suspect variables, denoted as the *error cardinality*, is constrained to a given constant N using cardinality constraints $\Phi_N(E)$ which can be generated from a network of adders [10]. Given an error trace $\mathcal{V}_0^k$ or a window of an error trace $\mathcal{V}_p^q$, design debugging can encoded by the following SAT problems respectively:

$$\begin{aligned} Debug_0^k &= S^0(s^0) \wedge \Phi_N(E) \wedge \\ &\quad \Big(\bigwedge_{i=0}^{k} X^i(x^i) \wedge Y^i(y^i) \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, E) \Big) \\ Debug_p^q &= S^p(s^p) \wedge \Phi_N(E) \wedge \\ &\quad \Big(\bigwedge_{i=p}^{q} X^i(x^i) \wedge Y^i(y^i) \wedge T_{en}(s^i, s^{i+1}, x^i, y^i, E) \Big) \end{aligned} \tag{2}$$

Note that for $N = 0$, $Debug_0^k$ is UNSAT, since the error trace applied to the erroneous design without any active error suspect variables cannot produce the correct outputs defined in the error trace.

In a satisfying assignment of Equation 2, each active suspect variable corresponds to a possible component (gate, module etc.) whose function can be changed to correct the erroneous behavior. To find all such suspects, for each satisfying assignment, a *blocking clause* is added to the debugging instance to block the active suspect variables from appearing again as a satisfying assignment. This instance is then sent again to the solver. When the solver eventually returns UNSAT, all possible suspects have been found.

Example 1 *Figure 1 shows a two time-frame expanded circuit of an erroneous two gate design with one state element. The suspect variables $\{e_1, e_2\}$ are denoted as enables on the side of each gate. The incorrect gate is g_2 which should be a buffer instead of an inverter. The error trace:*

$$\mathcal{V}_0^1 = \langle \overline{s_0^0}, \langle x_1^0 \wedge x_2^0, x_1^1 \wedge x_2^1 \rangle, \langle y_1^1 \wedge y_2^1 \rangle \rangle$$

demonstrates an erroneous behavior of the circuit. For $N = 1$, a satisfying assignment for the suspect variables $\{e_1, e_2\}$ is $\overline{e_1} \wedge e_2$. Adding the blocking clause $\overline{e_2}$ to the problem causes it to be UNSAT. This implies that g_2 is potentially the only gate that can be modified to correct the erroneous behavior.

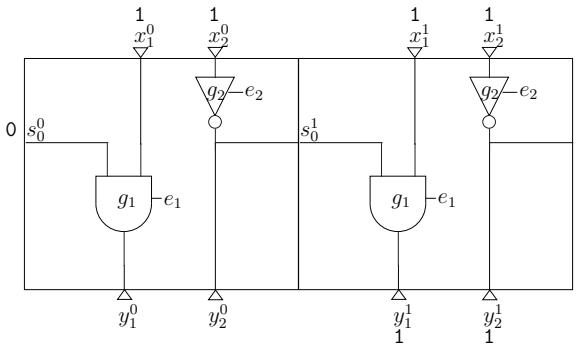


Fig. 1. SAT-based Debugging

C. Unsatisfiable Cores and Interpolants

An UNSAT core U is a subset of clauses that is unsatisfiable in an UNSAT propositional Boolean formula written in CNF. Modern DPLL [16] solvers can generate a proof of unsatisfiability along with a corresponding resolution graph that shows that a SAT instance is unsatisfiable [17]. The resolution graph demonstrates how clauses in the original SAT instance can be combined to generate the empty clause. The root nodes of the graph are the original clauses, the intermediate nodes correspond to the learned clauses and the leaf node is the empty clause.

An *interpolant* [18] is a Boolean formula that can be generated from an UNSAT core. For a given unsatisfiable formula whose clauses can be partitioned into two subsets, A and B, an interpolant is a formula, P, with the following properties:

(a) $A \rightarrow P$
(b) $B \wedge P$ is unsatisfiable.
(c) P only contains common variables of A and B.

There exists an algorithm [19] that can generate an interpolant as a Boolean circuit whose gates correspond to the vertices in the resolution graph and whose inputs correspond to the common variables. This algorithm takes $O(V+L)$ time, where V is the number of vertices in the resolution graph and L is the total number of literals in the proof. However, in the worst case, the size of the resolution graph can be exponential in the size of the problem.

III. Scalable Debugging with Interpolants

This section proposes a scalable SAT-based debugging algorithm that uses interpolants to reduce the number of simultaneous time-frames that need to be stored in memory. The algorithm analyzes windows of time-frames along the length of the error trace beginning with a suffix window and iteratively moving the window backwards until a prefix window of the error trace is analyzed. The interpolants are used to over-approximate constraints for a suffix of the error trace that is not modelled in the current window, ensuring that the erroneous behavior is properly constrained. Additionally, this method is shown to be complete and a technique using multiple interpolants is presented to improve its resolution.

In Section III-A and Section III-B, we show how to generate debugging instances for a suffix window and prefix window of an error trace. Using these two ideas, a complete scalable algorithm for debugging is described in Section III-C which partitions the original problem into smaller debugging instances. Finally, Section III-D shows how to improve resolution by using multiple interpolants.

A. Suffix Window Debugging

Debugging a suffix of an error trace can be achieved by applying the original SAT-based debugging scheme given in Equation 2. By using a suffix, only errors that are both excited within this window and propagate to primary outputs can be found. The following lemma describes a useful characteristic of suspects found in a suffix debugging instance.

Lemma 1 *Any suspect found in a debugging instance, $Debug_p^k$, for a suffix of an error trace, $\mathcal{V}_p^k$, will be found as a suspect to the debugging instance, $Debug_0^k$, for the entire error trace, $\mathcal{V}_0^k$.*

Proof: Let $M(E)$ be an assignment to the suspect variables in E such that $Debug_p^k \wedge M(E)$ is satisfiable. We wish to prove the lemma which can be written as:

$$Debug_p^k \wedge M(E) \text{ is SAT} \rightarrow Debug_0^k \wedge M(E) \text{ is SAT}$$

From Equation 2, we know that $Debug_0^{p-1} \wedge Debug_p^k$ and $Debug_0^k \wedge S^p(s^p)$ generate the same clauses. $Debug_0^{p-1}$ is SAT regardless of the error trace because the error has not been observed yet, so there is no mismatch in primary outputs. $Debug_0^{p-1} \wedge S^p(s^p)$ is SAT when no suspect variables are active because the instance $Debug_0^{p-1}$ amounts to simulating the circuit for the first p cycles of the error trace generating the same values as $S^p(s^p)$. Finally, $Debug_0^{p-1} \wedge S^p(s^p) \wedge M(E)$ is SAT because each active suspect variable allows the corresponding component to become an arbitrary non-deterministic function, which will not change the satisfiability of an instance if it was already satisfiable.

Therefore, if $Debug_p^k \wedge M(E)$ is SAT then $Debug_0^{p-1} \wedge Debug_p^k \wedge M(E)$ is SAT, since the only common variables are s^p and E which are fully assigned. As a result, $Debug_0^k \wedge S^p(s^p) \wedge M(E)$ is SAT implying that $Debug_0^k \wedge M(E)$ is SAT as required. ■

Lemma 1 guarantees that suspects found in the suffix are suspects that will be found in the entire error trace. However, if the error is excited before the current suffix, then there is no guarantee that the error will be found in $Debug_p^k$. Even though analyzing a suffix of an error trace may not result in a complete algorithm, valuable information can be extracted from the resulting UNSAT core as stated in the following theorem.

Theorem 1 *Let U be an UNSAT core generated after blocking all satisfying assignments to suspects for $Debug_p^k$. If $U \cap S^p(s^p) = \emptyset$ then the suspects found in $Debug_p^k$ will be exactly the suspects found in the entire debugging instance, $Debug_0^k$.*

Proof: From Lemma 1, any suspect found in $Debug_p^k$ is a suspect found in the entire debugging instance, $Debug_0^k$.

Now we prove by contradiction that any suspect found in $Debug_0^k$ will be found in $Debug_p^k$. Assume towards a contradiction that, $M(E)$ is an assignment to the suspect variables

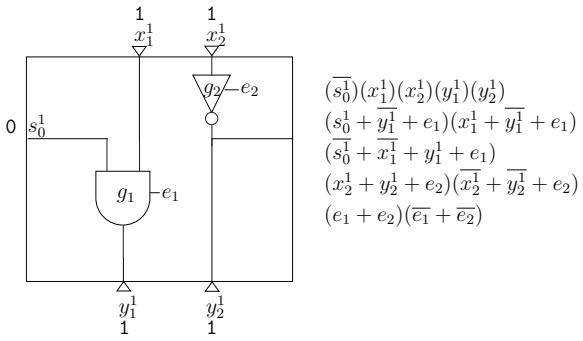


Fig. 2. Suffix Window Debugging

such that $Debug_0^k \wedge M(E)$ is SAT and $Debug_p^k \wedge M(E)$ is UNSAT. And let U be the UNSAT core derived after blocking all satisfying assignments to suspects for $Debug_p^k$, which contains no clauses in $S^p(s^p)$.

Since $Debug_p^k \wedge M(E)$ is UNSAT, $M(E)$ is not blocked by any of the blocking clauses to $Debug_p^k$, which we denote by $blocking\_clauses_p^k$. This means that $Debug_0^k \wedge blocking\_clauses_p^k$ is satisfiable. However we know that in terms of clauses, $U \subseteq (Debug_p^k \wedge blocking\_clauses_p^k - S^p(s^p)) \subseteq Debug_0^k \wedge blocking\_clauses_p^k$, since U does not contain any clauses from $S^p(s^p)$. However, $Debug_0^k \wedge blocking\_clauses_p^k \wedge M(E)$ is satisfiable, so $U \wedge M(E)$ is satisfiable. But U is an UNSAT core, which is a contradiction. So it must be the case that $Debug_p^k \wedge M(E)$ is SAT. ■

Theorem 1 gives a condition to omit a prefix debugging analysis with very little additional computation beyond extracting the UNSAT core from the suffix debugging instance. Notice that the proof does not depend on the error cardinality since the same UNSAT core will exist in the suffix instance as well as the entire debugging instance. However, in the case where Theorem 1 is not valid, the prefix of the error trace must be analyzed to get a complete set of suspects. The following example illustrates a case where Theorem 1 can not be applied because there are clauses in the UNSAT core from the initial state predicate.

Example 2 *A suffix debugging instance derived from Example 1 is shown in Figure 2. The suffix, $\mathcal{V}_1^1$, is used to produce a suffix debugging instance $Debug_1^1$ with $N = 1$. The clauses for the suffix debugging instance are shown to the right of the circuit diagram. This instance is unsatisfiable. The following is an UNSAT core from the instance:*

$$(x_2^1)(y_1^1)(y_2^1)(\overline{s_0^1})(\overline{e_1} + \overline{e_2})(\overline{x_2^1} + \overline{y_2^1} + e_2)(s_0^1 + \overline{y_1^1} + e_1)$$

Using Theorem 1, we know that $Debug_1^1$ does not result in the complete set of suspects to $Debug_0^1$ because the UNSAT core contains the clause $\overline{s_0^1} \subseteq S^1(s^1)$, so the prefix of the error trace still needs to be analyzed.

B. Prefix Window Debugging

Debugging a prefix of an error trace can be formulated in two parts. The first part uses the conventional SAT-based formulation (Equation 2) using a prefix of the error trace. The second part is an interpolant approximating time-frames for the corresponding suffix of the error trace.

Recall that the erroneous behavior is only observed in the last time-frame. If only a prefix of the error trace is modelled then the instance will not be properly constrained with the erroneous behavior. To avoid this situation, the interpolant is used as an over-approximation for the constraints that model the corresponding suffix. This ensures that the prefix debugging instance is properly constrained.

The interpolant can be generated by using an UNSAT core of the solved suffix debugging instance. To generate the interpolant, a partition of $Debug_p^k \wedge blocking\_clauses$ is defined by partitioning the clauses into two sets A and B. Set A represents the clauses modelling the enhanced transition function from p to k along with the primary input and output predicates from the error trace. Set B represents the initial state predicate, the error cardinality constraints and the blocking clauses. The clauses forming $Debug_p^k \wedge blocking\_clauses$ can be separated into A and B as follows:

$$\begin{aligned} A =& \bigwedge_{i=p}^{k} X^i(x_i) \wedge Y^i(y_i) \wedge T(s^i, s^{i+1}, x^i, y^i, E) \\ B =& S^p(s^p) \wedge \Phi_N(E) \wedge blocking\_clauses \end{aligned} \quad (3)$$

The common variables of A and B are the state variables s^p and the suspect variables E. Using this partition, an interpolant for the suffix, denoted P_p^k, can be generated from the resolution graph using the algorithm from [19].

P_p^k can be interpreted as an *over-approximation* of the suffix debugging instance. P_p^k will involve a subset of state and suspect variables that are directly related to the erroneous behavior observed at the primary outputs. The benefit of P_p^k is that it retains only the useful information that causes the erroneous behavior instead of modelling all the time-frames for the suffix of the error trace. In cases where the interpolant gets too large, the original clauses can be used in place of the interpolant, bounding the size of the constraints used to model the erroneous behavior. However experimental results show that in most cases, the interpolant is much smaller than the instance it was generated from, confirming the efficacy of using interpolants for debugging.

Example 3 *Figure 3 shows the resulting resolution graph on the left and interpolant on the right from the UNSAT core in Example 2. Notice how many of the root nodes of the resolution graph generate constants values in the interpolant. This is a common occurrence and generally leads to a small interpolant relative to the UNSAT core that it was derived from.*

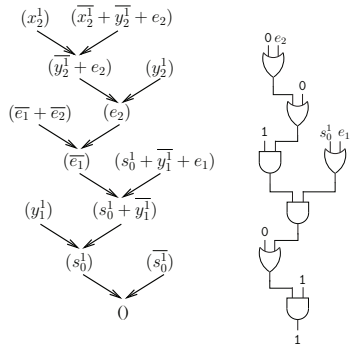


Fig. 3. Resolution graph and Interpolant

Using P_p^k, $Debug_0^{p-1}$ can be constrained with the cause for the erroneous behavior. The interpolant ensures that any suspect found in the prefix debugging instance resolves the erroneous behavior from the UNSAT core. The debugging instance for a prefix of an error trace with an interpolant, which we denote as $DebugItp_0^{p-1}$, can be written as follows:

$$DebugItp_0^{p-1} = Debug_0^{p-1} \wedge P_p^k \tag{4}$$

$DebugItp_0^{p-1}$ will be UNSAT when no suspect variables are active because $Debug_0^{p-1}$ will be equivalent to simulating the design for clock cycles 0 to $p-1$ and will implicitly generate the initial state predicate S^p which is known to be UNSAT with P_p^k. The next example builds from previous ones to show how a prefix debugging instance can be created.

Example 4 *Figure 4 shows how the interpolant generated in Example 3 can be used to debug a prefix of an error trace. Notice that the interpolant is significantly smaller once the constants have been propagated through the gates. In Figure 4, activating suspect variable, e_2, leads to the only satisfying assignment. This is consistent with the solution found in Example 1.*

The interpolant constrains the prefix debugging instance but it is an over-approximation. In other words, it will not miss suspects, as stated in the next theorem.

Theorem 2 *Any suspect found in $Debug_0^k$ will be found in $DebugItp_0^{p-1}$.*

Proof: By definition, $Debug_0^k = Debug_0^{p-1} \wedge A$, where A is defined in Equation 3. So any satisfying assignment to $Debug_0^k$ will satisfy $Debug_0^{p-1}$ and A. But $A \rightarrow P_p^k$, so it also satisfies P_p^k satisfying $DebugItp_0^{p-1}$. ■

Theorem 2 guarantees that solving the prefix debugging instance will result in a complete method where no suspects will be missed. However, it does not guarantee that spurious suspects will not be found. P_p^k is used as an over-approximation for the suffix, so it does not provide as many constraints as explicitly modelling time-frames p to k. This results in $DebugItp_0^{p-1}$ possibly returning suspects that will not be found when debugging the entire error trace. Section III-D aims to reduce these extra suspects and improve the resolution.

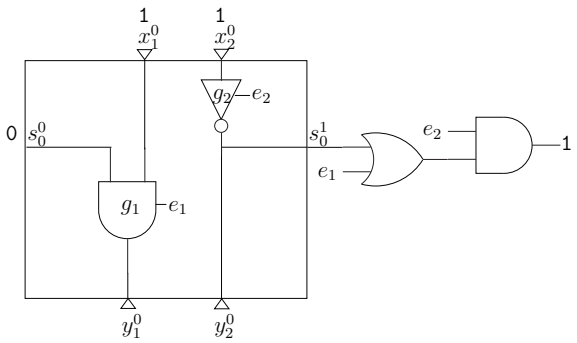


Fig. 4. Prefix Window Debugging with an Interpolant

Algorithm 1 Debugging with Interpolants

```
1:  step := maximum number of time-frames
2:  procedure DebugInterpolant(step)
3:      N := error cardinality
4:      E := set of potential suspect variables
5:      k := length of error trace
6:      solutions := suspects found by algorithm
7:      solutions ← ∅, P ← 1
8:      while k >= 0 do
9:          p ← max(k − step, 0)
10:         inst ← Debug_p^{k−1}(N, E) ∧ P
11:         solutions ← solutions ∪ SolveAll(inst)
12:         U ← ExtractUNSATCore(inst)
13:         if U ∩ S^p(s^p) = ∅ then
14:             return solutions
15:         end if
16:         P ← GenerateInterpolant(U)
17:         E ← E - solutions
18:         k ← k - step
19:     end while
20:     return solutions
21: end procedure
```

C. Scalable Debugging Algorithm

By using suffix and prefix debugging instances, it is possible to further divide the debugging problem into smaller windows that model no more than a user-defined number of time-frames. Algorithm 1 presents pseudo-code for a scalable debugging algorithm that divides an error trace of length k into $\lceil k/step \rceil$ windows, where $step$ is a user-defined parameter that specifies the maximum number of simultaneous time-frames to be modelled.

The algorithm iteratively analyzes windows of the error trace, starting with a suffix (lines 8-19). In each iteration, it begins by analyzing the current window of the error trace and finds all suspects shown on line 11. It then proceeds to generate an UNSAT core from the same instance and checks whether it contains any variables corresponding to the initial state predicate for the current instance. If it does not have any, it returns the current set of suspects. This is shown on lines 12-15. This condition allows for an early exit from the algorithm which in Theorem 1 was shown to be complete. If an early exit is not taken, it proceeds to generate an interpolant from the UNSAT core. Finally, it removes any suspects found in this iteration for consideration in the next iteration of the loop, pruning the search space for future iterations.

By iteratively analyzing consecutive windows of an error trace, the peak memory usage will be dramatically lowered with potential improvements in run-time. Even though the algorithm divides the error trace beyond just a suffix and prefix, it still guarantees completeness. This can be seen by analyzing each iteration of the loop. In the first iteration of the loop, if an early exit is taken (line 14), only a suffix debugging instance is run and Theorem 1 can be applied. After the first iteration of the loop for any given $step$, the prefix $\mathcal{V}_0^{k-step-1}$ of the error trace needs to be analyzed. This can be analyzed by using the debugging instance $DebugItp_0^{k-step-1}$ which is complete from Theorem 2.

However, instead of analyzing it directly, we can analyze a suffix of it by treating the interpolant as a constraint on the last time-frame. This is equivalent to another iteration of the loop. Using induction, this can be extended to the entire error trace and we can conclude that the last iteration of the loop will be a prefix debugging instance, $DebugItp_0^{k^*}$, where $V_0^{k^*}$ is the window used in the last iteration of the loop. By Theorem 2, this results in a complete algorithm.

Although Algorithm 1 is complete, there is a trade-off between the $step$ parameter and the final resolution. Each successive interpolant generated will potentially be a weaker constraint than the previous one. By setting $step$ to a small value, too many suspects can be returned. One way to cope with this is to provide a ranking of the suspects to the user so they can concentrate their effort on the most likely suspects. Algorithm 1 implicitly gives a useful ranking of suspects. More confidence can be given to suspects found in earlier iterations because a stronger constraint is used for the approximation of the suffix. In the case of the first iteration, all suspects found in the suffix will be found when debugging the entire error trace, as stated in Lemma 1.

D. Improving Resolution using Multiple Interpolants

The resolution of using this debugging method can be improved by using multiple UNSAT cores to generate multiple interpolants. Algorithm 1 guarantees completeness but may result in too many suspects if parameter $step$ is too small. This is due to the interpolant being an approximation to sets of constraints modelling the erroneous behavior. However by using multiple interpolants, the approximation will more closely match the original constraints potentially reducing the number of suspects that are found.

Using this fact, Algorithm 1 can be improved (line 12) by extracting multiple UNSAT cores to generate multiple interpolants. However, extracting multiple UNSAT cores can be an expensive process in general [20]. Algorithm 2 presents pseudo-code for a fast procedure for finding multiple UNSAT cores specifically for use in Algorithm 1.

The algorithm begins with an UNSAT instance and finds an UNSAT core (line 4). If the UNSAT core doesn't contain any clauses involving the initial state predicate, it exits and returns all UNSAT cores found so far (line 6-8). Otherwise, it randomly removes a subset of clauses from the initial state predicate that were involved in the current UNSAT core and is sent to the SAT solver again (line 10). This process is repeated until the instance is found to be satisfiable.

The size of the subset of initial state predicate clauses removed is a parameter to the algorithm. A smaller subset will leave more constraints in the problem having a higher chance of generating another UNSAT core but potentially taking more time and memory. By limiting the size of the subset and the number of cores found, the user can effectively trade-off run-time and memory for improved resolution.

Algorithm 2 Extracting Multiple UNSAT Cores

```
1:  procedure EXTRACTMULTIPLECORES(instance)
2:      Cores ← ∅
3:      while instance is UNSAT do
4:          U ← EXTRACTCORE(instance)
5:          CORES ← CORES ∪ {U}
6:          IF U ∩ S^p(s^p) = ∅ THEN
7:              RETURN CORES
8:          END IF
9:          to_remove ← SELECT_CLAUSES(S^p(s^p) ∩ U)
10:         instance ← instance − to_remove
11:     END WHILE
12:     RETURN CORES
13: END PROCEDURE
```

IV. EXPERIMENTS

This section presents experimental results for the proposed scalable SAT-based debugging algorithm as well as the algorithm to generate multiple interpolants. The results are compared to the SAT-based debugging work in [10] for the entire error trace which we will denote as *orig* in this section. MINISAT-v1.14 [21] with proof logging is used to solve the SAT instances and as well as generate the UNSAT cores. Experiments were run on a Pentium Core 2, 2.4 GHz workstation with 8GB of memory with a timeout of 7200 seconds.

We show the effectiveness of our algorithm on large designs from OpenCores.org [22]. Instances are generated by inserting a common RTL error such as a wrong assignment, missing case statement or incorrect operator. The error trace for each instance is generated by simulating the erroneous circuit through its testbench. Each suspect corresponds to a location in the RTL that can be corrected to satisfy the error trace.

Table I presents the results for the proposed debugging algorithm with interpolants. Four different sets of experiments are shown in this table. The first set of experiments in columns 5-7 correspond to running SAT-based debugging on the entire error trace (orig). The other three sets of experiments in column 8-16 correspond to debugging with interpolants varying the number of iterations ($r = \lceil k/step \rceil$) of the loop in Algorithm 1, ranging from 2 to 4. Each run uses one interpolant.

The first four columns in Table I show the instance name, number of clock cycles in the error trace, the gate count of the design and the total number of potential suspects. The next 12 columns show the run-time, peak memory and number of suspects returned for the four sets of experiments. For run-time and peak memory, the column with the lowest value is emphasized in bold.

For $r = 2$, the proposed algorithm shows on average a 24% decrease in run-time and 34% decrease in peak memory compared to orig, while increasing the number of suspects returned relative the total number of suspects on average by only 1%. With $r = 3$, the decrease in run-time is 26%, peak memory 48% and relative increase in suspects is 3%. $r = 4$ shows a similar trend by decreasing run-time by 23%, peak memory by 57%, but the relative increase in suspects is only 2% on average.

Figure 5 plots the run-time results from Table I from two different views. Figure 5(a) shows performance results of debugging with interpolants against orig on a log-log scale. Most points lie below the 45 degree line indicating faster runs on average. However, for several instances orig runs faster. In addition, `fdct1` and `fdct2` timed-out with orig while

TABLE I
DEBUGGING WITH INTERPOLANTS RESULTS

Instance Info				Orig			Interpolant, r=2			Interpolant, r=3			Interpolant, r=4		
instance	# cycles	# gates	total suspects	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols
ac971	675	25,314	1,086	588	6,040	34	357	2,842	34	253	2,022	34	**222**	**1,398**	34
ac972	300	25,314	1,086	314	2,674	41	133	1,187	41	112	838	47	**95**	**682**	47
divider1	40	5,799	1,092	10	180	32	**6**	127	41	6	110	41	6	**97**	41
divider2	40	5,799	1,092	5	188	21	**5**	121	21	5	109	21	6	**96**	38
fdct1	40	377,849	4,568	TIMEOUT			592	3,633	58	**412**	2,893	59	470	**2,437**	62
fdct2	40	377,849	4,568	TIMEOUT			851	4,819	54	460	2,889	54	**419**	**2,500**	57
fpu2	312	81,303	939	MEMOUT			295	6,704	4	206	4,692	4	**149**	**3,621**	4
fpu5	300	81,303	939	MEMOUT			841	7,764	34	**168**	4,448	44	810	**4200**	42
mem_ctrl1	100	46,425	2,451	174	2,901	12	150	1,655	12	94	1,187	12	**71**	**899**	12
mem_ctrl2	100	46,425	2,451	94	3,012	6	76	1,702	6	57	1,291	6	**44**	**944**	6
mrisc1	42	18,034	631	**31**	546	61	37	353	80	38	**305**	86	39	315	93
rsdecoder2	196	11,380	1,623	**20**	393	47	28	356	47	35	300	47	45	**245**	47
spi1	576	2,103	223	270	871	27	175	596	37	187	620	79	**84**	**338**	38
vga1	40	154,213	1,337	**203**	5,150	9	219	2,845	9	275	2,213	35	307	**1,832**	51
vga2	40	154,213	1,337	**383**	5,187	30	447	2,913	82	520	2,253	128	481	**1,637**	115
wb1	132	3,552	407	6	240	8	4	154	8	**3**	128	8	5	**122**	8
wb2	132	3,552	407	5	233	5	4	153	5	**3**	124	5	3	**120**	5

debugging with interpolants were able to successfully solve these instances.

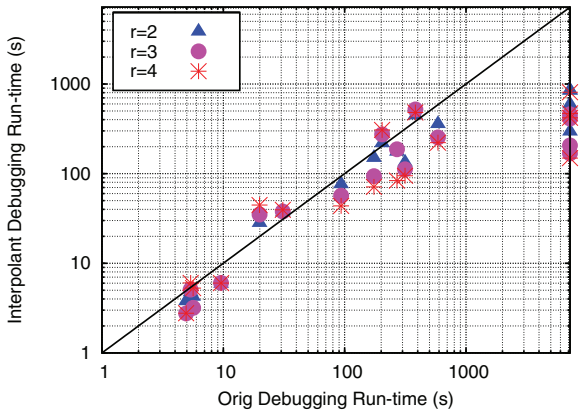
Taking a closer look at how run-time varies with the number of windows, r, Figure 5(b) shows how relative run-times of several designs vary with an increased r. The run-times are normalized to the orig instance indicated by $r = 1$. While most instances, show a reduction in run-time with larger r, `vga2` shows an increase. This can be attributed to the fact that the run-time of a debugging instance does not necessarily scale linearly with the problem size. However, most instances show a decrease in run-time as r is increased.

Figure 6 shows the benefit of using interpolants with respect to peak memory. Figure 6(a) shows the memory of using interpolants against orig on a log-log scale. All instances are below the 45 degree line indicating that they consistently require less memory.

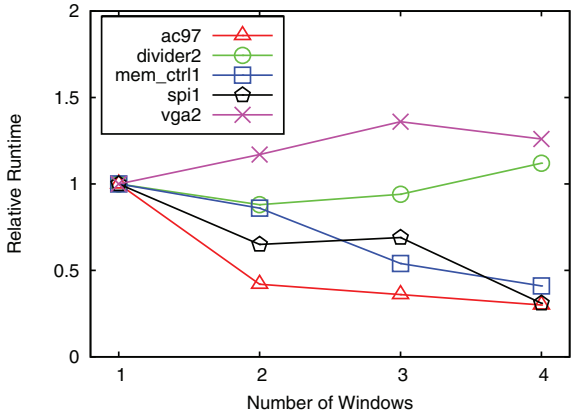
Looking more carefully at the relative memory usage for several instances, Figure 6(b) shows that the memory does not necessarily decrease inversely with r. `spi1` has a small relative increase from $r = 2$ to 3 but a much bigger relative decrease from $r = 3$ to 4. The reason for this is that the interpolant, which is not necessarily linear in size with the debugging instance, contributes to the peak memory. However, `ac972` shows a case where it does follow an inverse relation with r.

One would expect the early exit (line 14 from Algorithm 1) to contribute to this inverse relation. However, only the `mem_ctrl` and `wb` instances as well as `fpu2` used the early exit condition. This shows that the decrease in memory is due to the interpolant being significantly smaller than the instance it was generated from.

From Table I, we see that the number of suspects found generally increases as the r increases. This was explained in Section III-C due to potentially weaker interpolants being generated in later iterations of Algorithm 1. However, `spi1` and `vga2` show a case where the number of suspects actually decrease with increasing r. With $r = 3$ `spi1` generated 79 suspects while at $r = 4$ it generated 38. Similarly `vga2` had 128 at $r = 3$ and 115 at $r = 4$. These results suggest that although the interpolant is more likely to get weaker with increased r, how well it constrains the erroneous behavior can vary a great deal depending on the UNSAT core that was used.



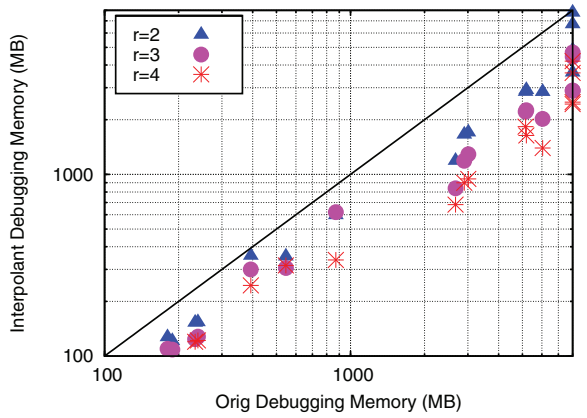
(a) Performance Scatter plot



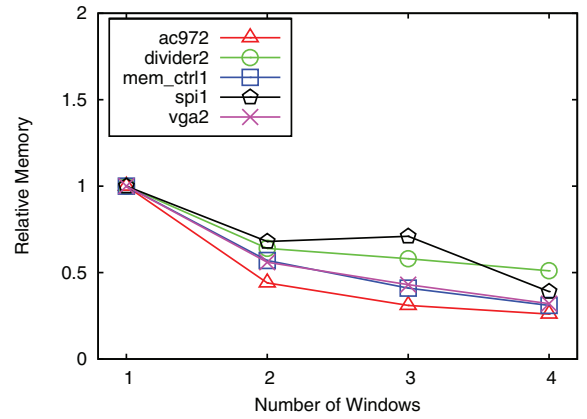
(b) Relative Performance Comparison

Fig. 5. Performance Results

Figure 7 shows the results from using multiple interpolants with $r = 4$ to constrain the debugging problem. The instances shown in this figure are ones where the number of suspects increased by a large amount over orig. For `spi1`, `vga1` and `vga2`, using multiple interpolants improved the quality of the debugging results by reducing the number of suspects.



(a) Memory Scatter plot



(b) Relative Memory Comparison

Fig. 6. Memory Results

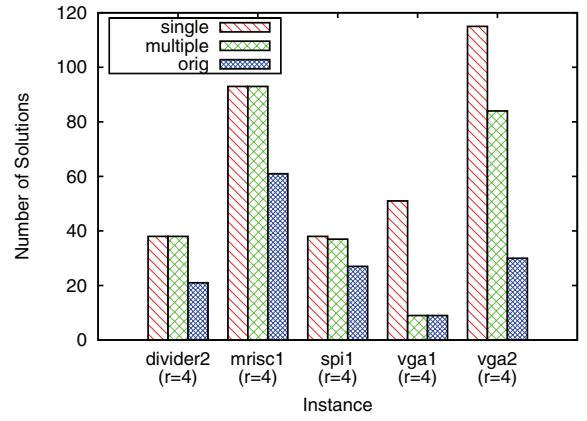


Fig. 7. Solutions using Multiple Interpolants

`vga1` shows a dramatic improvement in the number suspects returned where the interpolants help the suspects converge to the same value as orig. However, in some cases such as `divider2` and `mrisc`, multiple interpolants did not help constrain the problem further. These results show that the effectiveness of multiple interpolants is highly dependent on the debugging instance, where in some cases it can dramatically improve the resolution, while in others it does not help.

V. Conclusion

In this work, a scalable design debugging algorithm using interpolants is proposed. It partitions the problem into a sequence of smaller sub-problems that are easier to solve. Interpolants are used to reduce the number of simultaneous time-frames examined in the error trace by replacing sets of original clauses with a succinct approximation. The method is proven to be complete and an additional technique is presented to improve the quality of the debugging results using multiple interpolants. Experimental results show a large reduction in peak memory and improvements to run-time. This work encourages future research in design debugging using UNSAT cores and interpolation.

References

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, 2003.

[2] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 118–149, 2003.

[3] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *Int'l Conf. on CAD*, 2003, pp. 142–145.

[4] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in CAD*, 2004, pp. 159–173.

[5] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based unbounded symbolic model checking using circuit cofactoring," in *Int'l Conf. on CAD*, 2004, pp. 510–517.

[6] K.-H. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD*, 2005, pp. 1045–1051.

[7] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.

[8] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[9] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[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] S.Safarpour, M.Liffton, H.Mangassarian, A.Veneris, and K.A.Sakallah, "Improved design debugging using maximum satisfiability," in *Formal Methods in CAD*, 2007.

[12] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes Symp. VLSI*, 2008.

[13] H. Mangassarian, A.Veneris, S.Safarpour, M.Benedetti, and D.Smith, "A performance-driven QBF-based on iterative logic array representation with applications to verification, debug and test," in *Int'l Conf. on CAD*, 2007.

[14] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "Backspace: Formal analysis for post-silicon debug," in *Formal Methods in CAD*, 2008, pp. 1–10.

[15] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *IEEE Trans. on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.

[16] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Comm. of the ACM*, vol. 5, pp. 394–397, 1962.

[17] L. Zhang, "Searching for truth: Techniques for satisfiability of Boolean formulas," Ph.D. dissertation, Princeton, 2003.

[18] W. Craig, "Linear reasoning. a new form of the herbrand-gentzen theorem," *J. Symb. Log.*, vol. 22, no. 3, pp. 250–268, 1957.

[19] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, 2003.

[20] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[21] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[22] OpenCores.org, "http://www.opencores.org," 2007.

Debugging Formal Specifications Using Simple Counterstrategies

Robert Könighofer, Georg Hofferek, and Roderick Bloem
Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology

***Abstract*—Deriving a formal specification from an informal design intent is an error-prone process. The resulting specification may be incomplete, unrealizable, or in conflict with the design intent. We propose a debugging method for incorrect specifications that does not need an implementation.**

We show that we can explain conflicts with the design intent by explaining unrealizability. Our approach for explaining unrealizability is based on counterstrategies. Since counterstrategies may be large, we propose several ways to simplify them. First, we simplify the specification itself by removing both requirements and variables that do not contribute to the problem. Second, we heuristically search for a countertrace, i.e., a single input trace that suffices to demonstrate unrealizability. Finally, we present the countertrace or the counterstrategy to the user in extensive form as a graph and implicitly as an interactive game. We present experimental results for specifications given as GR(1) formulas.

I. Introduction

Ideally, a formal specification is written before the implementation. The specification can then be implemented either manually or automatically [20], [14], [10], [19]. In this scenario, the specification must have the highest quality possible. The same holds if the specification is written as independent verification IP. This scenario occurs quite frequently, for instance for protocols [7]. Creating a formal specification is an error-prone process and it is hard to achieve a high quality [12], [18], [6], [9], [5]. First, a specification may be incomplete. Second, it may be unrealizable, i.e., there may not be any implementation fulfilling it. Third, the specification might define something different from the engineer's original (informal) design intent. The aim of this paper is to debug specifications in the absence of an implementation.

Incomplete specifications have been addressed before. Katz et al. [12] analyze the completeness of a specification with respect to a given implementation. Claessen [6] gives a coverage analysis of a list of safety properties, introducing the notion of "forgotten cases" in which the system is underspecified. Fisman et al. [8] propose to check if a specification can be mutated into a simpler equivalent one in order to detect "inherent vacuity". The approaches of [6] and [8] are independent of an actual implementation.

The first contribution of this paper is a method to tackle inconsistencies between the specification and the design intent. Suppose there is an implementation that was either automatically synthesized from the formal specification or implemented manually. Suppose further that this system exhibits some behavior that differs from the designer's original intent, although the system conforms to the specification. In that case, either the specification is incomplete, or there is an inconsistency between the (informal) design intent and the specification. We will show that explaining such inconsistencies can be reduced to explaining unrealizability of specifications.

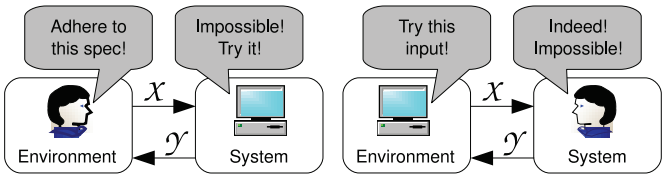
Unrealizability is a problem of its own. Our experience with the synthesis tools Lily [10] and Anzu [11] shows that mistakes during specification development often lead to unrealizability. Explaining unrealizability is difficult. There is no way to execute or simulate an unrealizable specification to track down the error, like one would do with erroneous implementations. Tools like RAT [18] explain why single traces do not fulfill the specification, but this does not suffice to explain unrealizability. Note that realizability is not the same as satisfiability. A specification is satisfiable if there is one input/output trace that satisfies the specification. In contrast, realizability requires that for each input trace we can construct a correct output trace step by step. Our case study shows that many unrealizable specifications are still satisfiable. Thus, known techniques from SAT solvers cannot be used to find the cause of unrealizability.

We present an interactive approach for explaining unrealizability, based on the following idea: When a user learns that her specification is unrealizable, she will be puzzled, since she must have imagined an implementation. In order to show that the imagined implementation is flawed, the debugging tool takes on the role of the environment, while the user takes on the role of the system. (See Fig. 1.) The tool provides inputs and the user tries to provide outputs conforming to the specification. The tool uses a counterstrategy to find inputs in such a way that there is no response of the system that fulfills the specification. Hence, the user will fail. However, while trying, she gains insights into why there is no way for her to comply with the specification, i.e., why the specification is unrealizable. She can subsequently use this knowledge to correct the specification.

Our experience shows that just presenting a counterstrategy does not suffice to explain unrealizability of larger specifications. The counterstrategy can be so complex that the user is unable to learn where the specification is too restrictive to be realizable. Thus, we present several simplifications. We adopt the idea of Cimatti et al. [5] to compute an unrealizable core. As the second contribution of this paper, we improve

This work was supported in part by the European Commission through project COCONUT (FP7-2007-IST-1-217069).

978-1-4244-4966-8/09/$25.00 ©2009 IEEE



(a) Simulating the System (b) Debugging Unrealizability

Fig. 1. Swapping the roles to gain insight into the cause of unrealizability.

over [5] by removing unnecessary signals and using delta debugging [26], an efficient minimization algorithm. Removing signals is crucial: just computing an unrealizable core leads to a specification that is harder to explain, not easier. Our third contribution is to use *countertraces* where possible. A countertrace is a fixed input trace for which there is no output trace fulfilling the specification. Focusing on this single trace makes it much easier for the user to localize the problem. A countertrace does not always exist and even if one exists, its computation is expensive. Hence, we present a heuristic.

After introducing our (generic) debugging approach, as a final contribution we show how it can be applied to specifications given in *Generalized Reactivity(1)* (GR(1), for short) [19]. GR(1) is a subset of LTL that has enough expressive power to be used for real world problems [2], [3] while still offering efficient symbolic algorithms [19]. We have evaluated our concepts for this class of specifications by integrating them into the synthesis tool Anzu [11].

Counterstrategies as debugging aids were previously used in the context of restricted specifications for timed systems [25], [1], in the context of Live Sequence Charts [4], and as witnesses or counterexamples to branching-time logic formulas [23], [22]. More efficient algorithms for the latter are presented in [15], [16], [24]. These papers mention simplification only peripherally, by discussing user interface and usability issues. We focus on simplifying the counterstrategy itself in order to convey meaningful information to the user, which is not done in the papers mentioned. (But see [5], as discussed above.) We are not aware of any previous work on finding countertraces, which we consider the most practical tool towards understanding why a specification is unrealizable. Also, to the best of our knowledge, counterstrategies have not been used before to explain conflicts between formal specifications and informal design intents.

The rest of the paper is organized as follows. Section II will revisit definitions and establish some notation. Section III introduces our debugging approach and Section IV concretizes it for GR(1) specifications. Section V presents our evaluation results for GR(1) and Section VI concludes the article.

II. Preliminaries

A. Automata

A *(deterministic and complete) automaton* is a tuple $\mathcal{A} = (Q, \Sigma, T, q_0, \mathrm{Acc})$, where Q is a finite set of states, Σ is a finite alphabet, $T : Q \times \Sigma \to Q$ is a deterministic and complete transition function, $q_0 \in Q$ is the initial state, and $\mathrm{Acc} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$ is the acceptance condition. A *run* of the automaton A on an (infinite) word $\overline{\sigma} = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $\overline{r} = q_0q_1q_2 \ldots \in Q^\omega$ such that $q_{i+1} = T(q_i, \sigma_i)$ for all $i \geq 0$. The run is *accepting* iff $\mathrm{Acc}(\overline{r}) = \mathsf{true}$.

A *deterministic and complete Büchi word automaton (DBW)* is an automaton in which Acc is given as a set of states $F \subseteq Q$ such that $\mathrm{Acc}(\overline{r}) = \mathsf{true}$ iff $\inf(\overline{r}) \cap F \neq \emptyset$, where $\inf(\overline{r})$ is the set of states that occur infinitely often in $\overline{r}$.

In the following, we assume that $Q = 2^V$ for a set V of state bits, and that $\Sigma = 2^X \times 2^Y$ for a set X of Boolean input signals and a set Y of Boolean output signals. We write $\mathcal{X} = 2^X$ and $\mathcal{Y} = 2^Y$. For $\overline{x} = x_0x_1 \ldots \in \mathcal{X}^\omega$ and $\overline{y} = y_0y_1 \ldots \in \mathcal{Y}^\omega$ we define $\overline{x}||\overline{y} = (x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ as their combination.

B. Generalized Reactivity

Generalized Reactivity (1) specifications form a subset of Linear Temporal Logic (LTL) [19]. They specify the interaction between an environment (controlling the input variables X) and a system (controlling the output variables Y) and consist of two parts: *assumptions* and *guarantees*. The specification states that the system must fulfill all guarantees whenever the environment fulfills all assumptions.

A GR(1) specification consists of $m+n$ DBWs representing m environment assumptions and n system guarantees [19]. With $\mathcal{A}^e_i = (Q^e_i, \Sigma, T^e_i, q^e_{0,i}, F^e_i)$ we denote the DBWs representing environment assumptions. The DBWs representing system guarantees are denoted $\mathcal{A}^s_j = (Q^s_j, \Sigma, T^s_j, q^s_{0,j}, F^s_j)$. All DBWs share the alphabet Σ. Let $\mathcal{A}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathrm{Acc})$ be the product of all DBWs $\mathcal{A}^e_i$ and $\mathcal{A}^s_j$, where the state space is $Q = Q^e_1 \times \cdots \times Q^e_m \times Q^s_1 \times \cdots \times Q^s_n$, the transition function is $T\left((q^e_1, \ldots, q^s_n), \sigma\right) = \left(T^e_1\left(q^e_1, \sigma\right), \ldots, T^s_n\left(q^s_n, \sigma\right)\right)$, and the initial state is $q_0 = (q^e_{0,1}, \ldots, q^s_{0,n})$. Let $J^e_i = \{(q^e_1, \ldots, q^s_n) \mid q^e_i \in F^e_i\}$ be the set of all states of the product automaton $\mathcal{A}^{\mathrm{GR1}}$ that are accepting in $\mathcal{A}^e_i$. Similarly, let J^s_j be the set of all states of $\mathcal{A}^{\mathrm{GR1}}$ that are accepting in $\mathcal{A}^s_j$. The acceptance condition Acc is

$$\mathrm{Acc}(\overline{r}) \Leftrightarrow (\forall i : \inf(\overline{r}) \cap J^e_i \neq \emptyset) \rightarrow (\forall j : \inf(\overline{r}) \cap J^s_j \neq \emptyset).$$

Thus, a run of $\mathcal{A}^{\mathrm{GR1}}$ is accepting iff all sets J^s_j of accepting states of the system are visited infinitely often, or some set J^e_i of accepting states of the environment is visited only finitely often.

C. Games and Strategies

A *game* is a tuple $\mathcal{G} = (Q, \Sigma, T, q_0, \mathrm{Win})$, where Q, T, and q_0 are defined as for DBWs and $\mathrm{Win} : Q^\omega \to \{\mathsf{false}, \mathsf{true}\}$. The game is played by two players. A *play* of $\mathcal{G}$ is an infinite sequence of states $\overline{\pi} = q_0q_1q_2 \ldots \in Q^\omega$, where $q_{i+1} = T(q_i, \sigma_i)$ for $i \geq 0$. The letters $\sigma_i = (x_i, y_i)$ are successively chosen by the players: In each step Player 1 first chooses x_i, after which Player 2 chooses y_i. A play $\overline{\pi}$ is won by Player 1 iff $\mathrm{Win}(\overline{\pi}) = \mathsf{true}$. Otherwise it is lost for Player 1 and won for Player 2. Note that Player 1 cannot react

to Player 2 and thus acts like a Moore machine. In contrast, Player 2 acts like a Mealy machine.

For GR(1) games, finite memory strategies suffice [19]. A *(finite memory) strategy* for Player 1 in the game $\mathcal{G}$ is a tuple (Γ, γ_0, ρ), where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$, Γ is some (finite) set representing the memory, and $\gamma_0 \in \Gamma$ is the initial memory content. The relation ρ maps a state of the game and the memory content to a set of possible choices for the inputs and an updated memory content. We require that ρ is complete, i.e., $\forall q, \gamma\, \exists x, \gamma' : (q, \gamma, x, \gamma') \in \rho$. A play $\overline{\pi} = q_0 q_1 \ldots$ *conforms to a strategy* (Γ, γ_0, ρ), iff there is a sequence $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\omega$ and a sequence $\gamma_0 \gamma_1 \ldots \in \Gamma^\omega$ such that for all $i \geq 0$, $(q_i, \gamma_i, x_i, \gamma_{i+1}) \in \rho$ and $q_{i+1} = T(q_i, (x_i, y_i))$. A strategy is *winning* from a state $q \in Q$ iff all plays starting from q and conforming to the strategy are won by Player 1. The *winning region* $W \subseteq Q$ of Player 1 is the set of states for which a winning strategy for Player 1 exists. A *counterstrategy* is a winning strategy for Player 1 from q_0.

A *co-GR(1) game* $\mathcal{G}_{\text{env}}^{\text{GR1}}$ (where Player 1 is the environment and Player 2 is the system) can be constructed from the automaton $\mathcal{A}^{\text{GR1}}$. The winning condition of $\mathcal{G}_{\text{env}}^{\text{GR1}}$ is the complement of the GR(1) acceptance condition:

$$\text{Win}(\overline{r}) \Leftrightarrow (\forall i : \inf(\overline{r}) \cap J_i^e \neq \emptyset) \wedge (\exists j : \inf(\overline{r}) \cap J_j^s = \emptyset) \quad (1)$$

D. μ-Calculus

We will use the propositional μ-calculus [13] extended with a mixed-preimage operator MX, defined on sets of states of a game. Let Var be a set of variables each representing a specific subset of Q. The syntax of μ-calculus formulas is defined recursively: Every subset $S \subseteq Q$ and every variable $Y \in \text{Var}$ is a μ-calculus formula. If P, Q are μ-calculus formulas, so are $\neg P$, $P \cup Q$, and $P \cap Q$, with the expected semantics. Furthermore, for $Y \in \text{Var}$, $\mu Y\,.\,P(Y)$, $\nu Y\,.\,P(Y)$, and $\mathsf{MX}(Y)$ are μ-calculus formulas defined as

$$\mu Y.P(Y) = \bigcup_i Y_i, \quad \text{where } Y_0 = \emptyset \text{ and } Y_{i+1} = P(Y_i), \quad (2)$$

$$\nu Y.P(Y) = \bigcap_i Y_i, \quad \text{where } Y_0 = Q \text{ and } Y_{i+1} = P(Y_i),$$

$$\mathsf{MX}(P) = \{q \in Q \mid \exists x \in \mathcal{X} : \forall y \in \mathcal{Y} : T(q, (x, y)) \in P\}\,.$$

The expression $\mathsf{MX}(P)$ denotes the set of states from which the environment can force a play into a state of P in one step. The order of existential and universal quantification corresponds to the fact that Player 1 moves first. We also define $\mathsf{MX}_x(P) = \{q \in Q \mid \forall y \in \mathcal{Y} : T(q, (x, y)) \in P\}$, i.e., the set of all states from which the games moves to P if Player 1 chooses input x.

E. Delta Debugging

Delta debugging [26] is an algorithm for minimization problems. Let S be a set that fails some test, denoted by $test(S) = \text{✘}$. We assume that some subset of S is "responsible" for the failing test and that $test$ is monotonic: $(test(S) = \text{✘}) \rightarrow (\forall S'' \supseteq S : test(S'') = \text{✘})$. The algorithm finds a minimal subset $S' \subseteq S$ so that $test(S') = \text{✘}$. The algorithm is defined recursively: $ddmin(c) = d(c, 2)$, and

$$d(c, n) = \begin{cases} d(c_i, 2) & \text{if } \exists i : test(c_i) = \text{✘} \\ d(\overline{c_i}, \max(n\text{-}1, 2)) & \text{else if } \exists i : test(\overline{c_i}) = \text{✘} \\ d(c, \min(|c|, 2n)) & \text{else if } n < |c| \\ c & \text{otherwise,} \end{cases}$$

where $(c_1, \ldots, c_n)$ is a partition of c into n approximately equally-sized parts. The set $\overline{c_i}$ is defined as $c \setminus c_i$.

The intuition behind this algorithm is a *divide and conquer* approach. At first, the set c is split in two parts and both are subjected to the test. If one of them fails the test, i.e., still contains a problem, the algorithm proceeds recursively on this part only. If both parts pass the test, elements from both parts contribute to the problem. Thus, the granularity of the partition is doubled. Since not only the subsets themselves but also their complements are tested, larger sets are tried as well. Eventually the algorithm returns a minimal subset that fails the test.

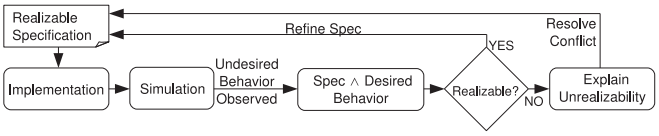


Fig. 2. The flow of our method to handle mismatches with the design intent.

III. Debugging Approach

This section introduces our generic debugging approach. We assume that we are given a temporal specification $\varphi = A \rightarrow G$ over X and Y, where A is a (possibly empty) set of environment assumptions and G is a set of system guarantees. We assume that we are given functions *realizable* and *sat* that decide realizability and satisfiability. We also assume that the specification can be turned into a two-player finite game and that we have a procedure that returns a finite state counterstrategy for an unrealizable specification. We further require that guarantees $g \in G$ can be removed from φ, and that output signals $y \in Y$ can be quantified existentially from the guarantees. These assumptions are relatively weak and hold for such logics as LTL. After presenting the general approach, we will instantiate it for GR(1).

A. Debugging Undesired Behavior

Inconsistencies between the formal specification and the designer's informal design intent may surface when a system has been built that satisfies the specification but not the design intent. The designer then has to change the specification to adhere to the design intent.

Fig. 2 illustrates our approach to handle mismatches with the design intent. We distinguish two cases:

1) The specification is incomplete, i.e., there is an implementation of the specification meeting the design intent.
2) Any system exhibiting the desired behavior violates the specification. This means that the specification is so

restrictive that there exists no implementation of the specification that shows the desired behavior.

The two cases can be distinguished by augmenting the specification with a guarantee that enforces the required behavior. If the modified specification is realizable, the original specification was incomplete and needs to be refined. Otherwise, we need to explain to the user why enforcing the desired behavior makes the specification unrealizable.

First, however, the desired behavior must be specified. We would like to provide the user with a method to obtain a relatively general requirement from the incorrect trace. We propose to allow the user to change any number of signal values in the incorrect trace to the desired value 0, 1, or $-$, where $-$ stands for "don't care". The tool then checks whether the input part of the trace conforms to the environment assumptions. If not, a corresponding warning is given to the user as the system is not required to fulfill its guarantees for such inputs. Next, the tool converts the given trace into a guarantee g_{new} in the following way. Each time step i of the desired trace $\bar{t}$ represents a function $t_i \colon (X \cup Y) \to \{0, 1, -\}$. An input trace $\overline{x} = x_0 x_1 \ldots \in \mathcal{X}^\omega$ conforms to the desired trace $\bar{t}$ (written $\overline{x} \overset{\mathcal{X}}{\models} \bar{t}$) iff

$$\forall i : \forall s \in X : (t_i(s) = 0 \rightarrow s \notin x_i) \wedge (t_i(s) = 1 \rightarrow s \in x_i).$$

Conformance with an output trace (written $\overline{y} \overset{\mathcal{Y}}{\models} \bar{t}$) is defined analogously. The guarantee g_{new} must accept exactly the words $(\overline{x}||\overline{y}) \in \Sigma^\omega$ such that $\overline{x} \overset{\mathcal{X}}{\models} \bar{t} \rightarrow \overline{y} \overset{\mathcal{Y}}{\models} \bar{t}$. This enforces the desired outputs whenever the given input scenario applies. The construction of g_{new} from the desired trace $\bar{t}$ depends on the actual specification language.

The guarantee g_{new} is then added to the original specification $\varphi = A \rightarrow G$ to obtain $\varphi' = A \rightarrow G \cup \{g_{\text{new}}\}$. Next, we check φ' for realizability. If φ' is realizable, it is a valid refinement of φ and the undesired behavior has been successfully eliminated. Otherwise we proceed by explaining the reasons for unrealizability as outlined in the next section.

B. Debugging Unrealizability

The flow of our method to explain unrealizability is depicted in Fig. 3. First, we check for satisfiability. For unsatisfiable specifications, trace-based debugging methods can be used [18]. If the specification is satisfiable, but not realizable, we apply a minimization step to find an unrealizable core. Only the part that is inconsistent or in conflict with the design intent (cf. Section III-A) remains. Based on this simplified specification, we then compute a counterstrategy. The counterstrategy is a finite state strategy such that the specification cannot be fulfilled if the environment adheres to it. We then attempt to obtain a countertrace from the counterstrategy. Countertraces and counterstrategies are finally presented to the user as a summarizing graph, and in form of an interactive game. The next sections explain the steps of this procedure in more detail.

C. Minimizing the Specification

Debugging an unrealizable specification $\varphi = A \rightarrow G$ is especially hard if it is large. However, the cause of unrealizability often involves only small parts of φ. Removing the rest

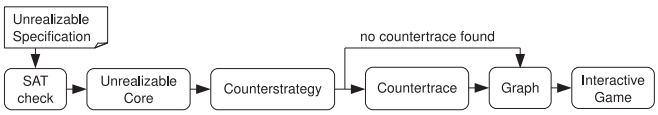


Fig. 3. The flow of our method to explain unrealizability.

leads to a simpler specification $\hat{\varphi}$ that is still unrealizable and easier to debug. Our experiments will show that minimization speeds up the computation of counterstrategies and leads to easier-to-understand games.

Removing environment assumptions would confuse the user during the interactive game as it adds (environment) behavior that the user originally excluded with these assumptions. Thus, we only remove system guarantees, looking for a minimal unrealizable core. Removing guarantees leads to shorter specifications. Surprisingly, the corresponding game is often more difficult to understand than the original. The reason is that removing guarantees adds (system) behavior. This results in more possible plays and a larger game graph. We counteract this effect by removing unnecessary outputs as well. We can do this by existentially quantifying them from all guarantees. This operation will be denoted $\exists Y' : G$ for some set $Y' \subseteq Y$ of outputs to remove.

Lemma 1: If $\varphi = A \rightarrow G$ is a realizable specification, so is $\varphi' = A \rightarrow (\exists Y' : G')$, for all $G' \subseteq G$, $Y' \subseteq Y$.

Lemma 2: Let (Γ, γ_0, ρ) be a counterstrategy for specification $\varphi' = A \rightarrow (\exists Y' : G')$, with $G' \subseteq G$, $Y' \subseteq Y$. Then (Γ, γ_0, ρ) is also a counterstrategy for $\varphi = A \rightarrow G$.

Lemma 1 states that removing guarantees or output signals preserves realizability. Furthermore, Lemma 2 states that minimizing the number of guarantees and output signals is helpful for finding a simple explanation for unrealizability, because the counterstrategy for the minimized system also applies to the original specification.

Cimatti et al. [5] propose to find an unrealizable core by removing one guarantee after the other. Thus, they require exactly $|G|$ checks for realizability to find the core. We attempt to reduce the average number of checks by using delta debugging [26]. This algorithm expects a set to be minimized as argument and uses a function $test$. We define $test(G' \cup Y') = realizable(A \rightarrow (\exists Y \setminus Y' : G'))$ and compute a minimized specification $\hat{\varphi} = A \rightarrow (\exists Y \setminus \hat{Y} : \hat{G})$, where $\hat{Y} = D \cap Y$, $\hat{G} = D \cap G$, and $D = ddmin(G \cup Y)$. We apply Lemma 1 to further reduce the number of realizability checks (cf. [26]): We store all sets $R = G' \cup Y'$ such that $A \rightarrow (\exists Y \setminus Y' : G')$ is realizable. We do not have to recompute realizability if we encounter a subset R' of a stored set R.

Theorem 1: $\forall G' \subseteq \hat{G}, Y' \subseteq \hat{Y} : ((G', Y') \neq (\hat{G}, \hat{Y})) \rightarrow realizable(A \rightarrow (\exists Y \setminus Y' : G'))$.

Proof: See the proof of Proposition 11 in [26]. ■

D. Countertraces

In general, the inputs given by the counterstrategy depend on the previous outputs of the system. Thus, a counterstrategy can be viewed as a graph or an interactive game. To make

things easier for the user we would prefer to construct a single *countertrace*, i.e., an infinite trace $\overline{x} \in \mathcal{X}^\omega$ for which there is no $\overline{y} \in \mathcal{Y}^\omega$ such that $\overline{x}||\overline{y}$ fulfills the specification. If the inputs that the user faces are always the same, regardless of her choice of outputs, the time and effort for understanding why she always loses the game is much lower. Unfortunately, such a countertrace does not always exist. A typical example is the LTL specification $\mathsf{G}(y \leftrightarrow \mathsf{X}\, x)$. This specification is not realizable, but for any given input trace $\overline{x}$ there is an output trace $\overline{y}$ such that $\overline{x}||\overline{y}$ fulfills the specification [21], [17].

We can compute whether a countertrace exists by existentially quantifying the output variables from the game automaton, complementing the automaton, and checking for emptiness. However, the game automaton may be non-deterministic after quantification and complementing it would cause an exponential blow-up. We consider that intractable and therefore present a heuristic. It does not always find a countertrace, even if one exists. However, our experiments show that it suffices for many cases of interest.

Our heuristic starts with a counterstrategy (Γ, γ_0, ρ) for a game $\mathcal{G} = (Q, \Sigma, T, q_0, \mathrm{Win})$, where $\rho \subseteq (Q \times \Gamma \times \mathcal{X} \times \Gamma)$. We simultaneously compute a countertrace $\overline{x} = x_0 x_1 \ldots \in \mathcal{X}^\omega$ and a sequence of sets $S_i \subseteq (Q \times \Gamma)$, where S_i contains all pairs of state and memory content possible after $x_0 \ldots x_{i-1}$ has been used as input. We start with $S_0 = \{(q_0, \gamma_0)\}$ and define

$$S_{i+1} = \big\{(q', \gamma') \;\big|\; \exists (q, \gamma) \in S_i, y \in \mathcal{Y} : \\ q' = T(q, (x_i, y)) \wedge (q, \gamma, x_i, \gamma') \in \rho\big\},$$

where x_i is chosen arbitrarily from the set $T_i = \{x \in \mathcal{X} \mid \forall (q, \gamma) \in S_i \, \exists \gamma' : (q, \gamma, x, \gamma') \in \rho\}$. Intuitively, the set T_i contains all inputs that conform to the counterstrategy, no matter in which precise state $(q_i, \gamma_i) \in S_i$ the play is. Thus, x_i is independent of the previous moves of the system. If $T_i = \emptyset$ for any i, our heuristic fails.

Let k be the smallest number such that $S_k \subseteq S_j$ for some $j < k$. We can easily show by induction that if we pick $x_{k+i} = x_{j+i}$, then $T_{k+i} \supseteq T_{j+i}$ and $\emptyset \subset S_{k+i} \subseteq S_{j+i}$ for all $i \geq 0$ (the former inclusion by completeness of T, the latter by monotonicity of the definition of S_{i+1}).

Thus, we can stop the computation when we find a set inclusion, obtaining a lasso-shaped countertrace $\overline{x}$ composed of the finite stem $x_0 \ldots x_{j-1}$ and infinite many repetitions of $x_j \ldots x_{k-1}$. Although the upper bound for k is exponential in the number of states, our experiments show that it is typically rather small (< 10 in most cases).

A countertrace represents a strategy $(\Gamma_{\overline{x}}, 0, \rho_{\overline{x}})$ with memory $\Gamma_{\overline{x}} = \{0, \ldots, k-1\}$. We define

$$\rho_{\overline{x}} = \big\{(q, i, x_a, i \oplus 1) \;\big|\; q \in Q \wedge i \in \Gamma_{\overline{x}}\big\}, \text{ where}$$
$$i \oplus 1 = \begin{cases} i+1 & \text{if } a < k-1, \\ j & \text{if } a = k-1. \end{cases}$$

Theorem 2: Every play $\overline{\pi}$ that conforms to the strategy $(\Gamma_{\overline{x}}, 0, \rho_{\overline{x}})$ also conforms to the counterstrategy (Γ, γ_0, ρ) and is thus won by the environment.

Proof: The inputs x_i dictated by $\rho_{\overline{x}}$ are (singleton) subsets of the inputs that are allowed by ρ. This follows trivially from the construction of $\overline{x}$. ∎

E. Graphs and Interactive Games

We suggest to use the counterstrategy (or the countertrace, if found) in two ways in order to illustrate the reason for unrealizability. First, the user can explore the counterstrategy in an interactive game. In every step, the strategy suggests an input and the user selects an output of the system. Playing and losing the game (repeatedly) allows the user to discover the reason for which the specification is unrealizable. Second, we display a graph $\mathbb{G}$ that summarizes all possible plays in the game. Vertices in this graph correspond to state-memory pairs $(q, \gamma) \in (Q \times \Gamma)$, edges represent transitions that are allowed. The graph can be seen as a "cheat sheet" for the interactive game: it allows the user to see how the environment will react to her outputs. Thus, she may discard some choices a priori and thereby reduce the number of plays necessary to understand the cause of unrealizability.

IV. Debugging GR(1) Specifications

In this section we concretize our generic debugging method for GR(1) specifications. The class of GR(1) fulfills all the necessary premises stated in III. The environment assumption as well as the system guarantees are each represented by a set of DBWs. Our approach for debugging undesired behavior requires that the desired behavior can be transformed into a guarantee. Constructing a DBW that accepts the desired behavior is trivial when it is given as a trace. Removing guarantees reduces to removing DBWs from the according set. Removing output signals is done by existentially quantifying them in the symbolic representation of the DBWs. Realizability can be decided as shown in [19]. Synthesis of a counterstrategy for an unrealizable GR(1) specification has not been addressed before in the literature. We will show how this can be achieved in the next section. Furthermore, we address some GR(1) specific aspects of its illustration.

A. Counterstrategies for GR(1) Specifications

We derive a counterstrategy for the co-GR(1) game $\mathcal{G}_{\mathrm{env}}^{\mathrm{GR1}} = (Q, \Sigma, T, q_0, \mathrm{Win})$ from some intermediate results in the calculation of the winning region for the environment (Player 1). The winning region for the system (Player 2) is defined in [19]. The winning region for the environment is its complement. Hence, we obtain

$$W_{\mathrm{env}}^{\mathrm{GR1}} = \mu Z \,.\, \bigcup_{j=1}^{n} \nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,. \\ \big(\neg J_j^s \cup \mathsf{MX}\, Z\big) \cap \mathsf{MX}\, Y \cap (J_i^e \cup \mathsf{MX}\, X)\,. \quad (3)$$

Theorem 3: The set $W_{\mathrm{env}}^{\mathrm{GR1}}$ is the set of winning states for the environment in the co-GR(1) game $\mathcal{G}_{\mathrm{env}}^{\mathrm{GR1}}$.

In order to formulate a counterstrategy, we define Z_a to be the a-th iteration (according to Equation 2) of the fixpoint

computation of Z in Equation 3. We also define $Y_{a,j}$ as

$$\nu Y \,.\, \bigcap_{i=1}^{m} \mu X \,.\, \left(\neg J_j^s \cup \mathsf{MX}\, Z_{a-1}\right) \cap \mathsf{MX}\, Y \cap \left(J_i^e \cup \mathsf{MX}\, X\right).$$

Finally, $X_{a,j,i,c}$ is the c-th iteration of the fixpoint computation

$$\mu X \,.\, \left(\neg J_j^s \cup \mathsf{MX}\, Z_{a-1}\right) \cap \mathsf{MX}\, Y_{a-1,j} \cap \left(J_i^e \cup \mathsf{MX}\, X\right).$$

To ease notation, we also define $Z_a^{\text{new}} = Z_a \setminus Z_{a-1}$ and $X_{a,j,i,c}^{\text{new}} = X_{a,j,i,c} \setminus X_{a,j,i,c-1}$. We will further write $i \oplus 1$ for $(i \mod m) + 1$.

Let $\hat{Q} = Q \cap W_{\text{env}}^{\text{GR1}}$ be the set of states from which a counterstrategy for the co-GR(1) game exists. To obtain such a counterstrategy, we define four sub-strategies $\rho_1, \rho_2, \rho_3, \rho_4 \subseteq (\hat{Q} \times \Gamma \times \mathcal{X} \times \Gamma)$, where $\Gamma = \mathcal{I} \times \mathcal{J}$. The set $\mathcal{I} = \{1, \ldots, m\}$ stores the index of the next set J_i^e of accepting states of the environment that the play will reach. The set $\mathcal{J} = \{0, 1, \ldots, n\}$ stores the index of the set J_j^s of accepting states of the system that the environment tries to evade. The value 0 is added to store the fact that the environment has not (yet) committed to any such set. The initial memory content is $\gamma_0 = (1, 0)$.

We will build a strategy that makes sure that the play never moves from an iterate Z_i^{new} to an iterate $Z_{i'}^{\text{new}}$ with $i' > i$. Furthermore, for each i, there is a j such that if the play remains in Z_i^{new} then J_j is never visited. This j is stored in the second element of the memory. It is then easy to see that the environment always wins: If the play reaches Z_1, it is trapped, and the environment wins. In a higher iterate, either a system fairness condition is never fulfilled, or the game moves to a lower iterate. It is important that the environment takes advantage of each opportunity to take the play into a lower iterate, for otherwise the system could be able to visit J_j^s states infinitely often for all j.

Sub-strategy ρ_1 is used to take the game into a smaller iterate of Z whenever possible. This step changes the value of j. However, the system's choice of y can influence to which $Y_{a-1,j}$ the play proceeds. Thus, the environment can only choose the new value for j after the system's move. In order to remember to do so, j is set to 0:

$$\rho_1 = \left\{(q, (i, j), x, (i, 0)) \;\middle|\; \exists\, a \geq 2 : q \in Z_a^{\text{new}} \cap \mathsf{MX}_x(Z_{a-1})\right\}$$

Sub-strategy ρ_2 is applied whenever $j = 0$. It sets j to a suitable value depending on the state the play is in:

$$\rho_2 = \left\{(q, (i, 0), x, (i, j')) \;\middle|\; \exists\, a \geq 1 : q \in (Z_a^{\text{new}} \cap \mathsf{MX}_x(Y_{a,j'})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

Sub-strategy ρ_3 is applied if the play is in a state of J_i^e. A state in $J_{i\oplus 1}^e$ should be reached next (possibly in several steps), thus ρ_3 updates the content of i:

$$\rho_3 = \left\{(q, (i, j), x, (i \oplus 1, j)) \;\middle|\; j \neq 0 \;\wedge\; q \in J_i^e \;\wedge\; \exists\, a \geq 1 : q \in (Z_a^{\text{new}} \cap \mathsf{MX}_x(Y_{a,j})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

Sub-strategy ρ_4 is used when the set J_i^e is not yet reached. It is an attractor strategy forcing the play ever closer to J_i^e:

$$\rho_4 = \left\{(q, (i, j), x, (i, j)) \;\middle|\; j \neq 0 \;\wedge \exists\, a \geq 1, c \geq 2 : (q \in Z_a^{\text{new}} \cap X_{a,j,i,c}^{\text{new}} \cap \mathsf{MX}_x(X_{a,j,i,c-1})) \setminus \mathsf{MX}(Z_{a-1})\right\}$$

Theorem 4: In the co-GR(1) game $\mathcal{G}_{\text{env}}^{\text{GR1}}$, the strategy $\left(\mathcal{I} \times \mathcal{J}, (1, 0), \rho_{\text{env}}^{\text{GR1}}\right)$ with $\rho_{\text{env}}^{\text{GR1}} = \rho_1 \cup \rho_2 \cup \rho_3 \cup \rho_4$ is a counterstrategy.

B. Graphs and Interactive Games in case of GR(1)

The current memory content of the counterstrategy is presented to the user during the interactive game as well as in the graph $\mathbb{G}$. Knowing the index j of the set J_j^s which the environment tries to evade, the user can focus on reaching this set only. The user might indeed be able to reach a J_j^s state. However, by doing so she allows the tool to force the play into a smaller iterate of Z (using ρ_1).

V. Experimental Results

Our implementation, the specifications, and the scripts needed to reproduce the evaluations are available for download on the Anzu website[1]. Our results are summarized in Table I. All experiments were performed on an Intel Centrino 2 processor with 2×2.0 GHz and 3 GB RAM. We used two different specifications, both parametrized. The first one defines a bus arbiter [2], parameterized with the number of masters. We denote its variants by `Axy`, where `x` is the number of masters and `y` is the kind of error we introduced in order to make the specification unrealizable. With `woef` we indicate that a fairness constraint of the environment was removed, `wsf` means that a fairness constraint of the system was added, and `wst` means that we impose additional restrictions on state transitions of the system. The second specification defines a generalized buffer [3] used by n senders and two receivers. We denote its variants by `Gxy`, where `x` defines the number of senders and `y` is as before. All specification variants are satisfiable but not realizable. Their size ranges from 90 properties over 22 signals (`A2woef`) to 6004 properties over 218 signals (`G100wst`).

Columns 1 to 4 present results without minimization. Columns 1 and 2 list the times for computation of the winning region $W_{\text{env}}^{\text{GR1}}$ and the counterstrategy's relation $\rho_{\text{env}}^{\text{GR1}}$. Column 3 shows the number of vertices in the graph $\mathbb{G}$. Column 4 indicates if a countertrace was found. Columns 5 to 10 summarize results when delta debugging is applied. Column 5 gives the number of realizability checks and Column 6 relates this to the number of checks necessary with the algorithm of [5], which we reimplemented for comparison. Column 7 gives the time for minimization with the algorithm of [5]. The time needed for delta debugging is shown in Column 8, and the time savings compared to [5] are shown as speed-up factor in Column 9. Column 10 finally contains the number of vertices in $\mathbb{G}$ for the minimized specification. This number is a good

[1] http://www.iaik.tugraz.at/content/research/design_verification/anzu/

TABLE I
PERFORMANCE RESULTS. DD = "DELTA DEBUGGING".

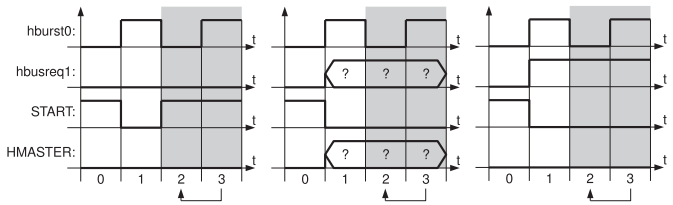
column	1	2	3	4	5	6	7	8	9	10
	without DD				with DD					
	Time: $W_{\text{env}}^{\text{GR1}}$	Time: $\rho_{\text{env}}^{\text{GR1}}$	# Vertices in $\mathbb{G}$	$\not\vdash$ found	# Checks during DD	Reduction of Checks	Time: $\hat{\varphi}$ in [5]	Time: $\hat{\varphi}$ with DD	Speed-Up Factor	# Vertices in $\mathbb{G}$
	[sec]	[sec]	[-]	[-]	[-]	[%]	[sec]	[sec]	[-]	[-]
A2woef	0.3	0.6	27	yes	47	41	5.3	0.7	7.6	5
A4woef	30	23	75	yes	56	59	284	5.1	56	13
A6woef	463	325	267	yes	58	70	7431	33	225	29
A2wsf	0.6	0.4	59	yes	37	54	8.7	0.7	12	5
A4wsf	38	22	171	yes	46	66	754	9.7	78	5
A6wsf	683	248	715	yes	47	76	6958	18	387	5
A2wst	0.4	0.3	43	yes	41	49	4.4	0.8	5.5	7
A4wst	10	18	139	yes	52	62	260	6.6	39	19
A6wst	644	410	683	yes	55	71	11170	72	155	43
G10wsf	2.1	1.8	50	no	124	19	65	53	1.2	9
G20wsf	0.8	2.4	92	no	215	42	29	118	0.3	9
G40wsf	2.0	12	176	no	473	57	165	1317	0.1	9
G100wsf	10	225	204	no			4205	>40k		
G10wst	0.2	0.2	10	yes	42	73	3.4	1.1	3.1	7
G20wst	0.5	0.6	22	yes	40	89	13	2.0	6.5	7
G40wst	2.7	4.3	40	yes	61	94	105	4.0	26	7
G100wsf	19	64	46	yes	64	99	4214	18	234	7
total	1897	1133					31470	1660	19	

indicator how simple it is to gain meaningful insights. As it can be seen, it is reduced dramatically by minimization.

In our experiments, a countertrace was found in about 80% of the cases without minimization, and in all cases after minimization. Our subjective impression is that countertraces are far easier to understand than strategies. The time for the computation of the countertrace from a given counterstrategy is negligible. Hence, with very little additional computational costs, our heuristic provides us with much simpler explanations for unrealizability in many cases.

Minimization reduces the number of formulas in the specification by 85% on average, making them much easier to analyze. Simultaneously, it raises the chance that a countertrace is found and significantly reduces the size of the graph that the user must explore, if no trace is found. Minimization does not take much time in most cases. Additionally, the times for the computation of counterstrategies, countertraces and graphs for the simplified specification are then negligible. Thus, using minimization provides results that are more helpful for the user, without additional costs in terms of CPU time in many cases.

Compared to [5], delta debugging needs about 50% fewer realizability checks in average. However, the reduction in the number of checks does not directly correlate with the time savings, as the time per check heavily depends on the complexity of the specification. Following [5], this complexity decreases quite steadily. With delta debugging there is a chance of removing many guarantees in one step. If this happens early, the performance of delta debugging is overwhelmingly superior. If significant reductions happen late, the performance may be worse, which explains the results for `G20wsf`, `G40wsf`, and `G100wsf`. However, in most cases of our experiments delta debugging is much faster. Both minimization methods lead to specifications of about the same size.



(a) Simulation (b) Design Intent (c) Countertrace

Fig. 4. Example: Illustrating a conflict with the design intent.

Example

In this section we present a practical example that illustrates our concepts. It specifies a bus arbiter with 2 masters. We will use lower case letters for inputs and upper case letters for outputs. The output signal `HMASTER` is set to 0 whenever the bus is currently owned by master 0, and set to 1 whenever the bus is occupied by master 1. The output `START` must be raised when the bus ownership changes. With the inputs `hbusreq0` and `hbusreq1`, the bus can be requested by master 0 and master 1, respectively. There are further signals, among them `hburst0`, but their meaning is irrelevant for this example.

Fig. 4(a) depicts a possible simulation run, simplified to signals of interest. The infinite loop is marked with gray background. Suppose the design intent was that `START=0` as long as the input `hburst0` keeps on changing. This behavior is not observed. Following our debugging approach (see Section III-A), the user specifies the design intent as shown in Fig. 4(b). This design intent is added as an additional guarantee to the specification. The new specification (A2wst in Table I) is unrealizable, which means that the design intent that was just added is in conflict with the rest of the specification.

To explain this conflict, our tool first minimizes the specification. The minimization algorithm removes 70% of the 15 output signals and 90% of the 66 formulas specifying the system. Besides the trace with which the user specified the design intent, the following guarantees remain:

$$\texttt{HMASTER=0} \wedge \texttt{START=1}, \tag{4}$$

$$\mathsf{G}((\mathsf{X}\,\texttt{START=0}) \rightarrow (\texttt{HMASTER=1} \leftrightarrow \mathsf{X}\,\texttt{HMASTER=1})) \tag{5}$$

$$\mathsf{GF}(\texttt{HMASTER=1} \vee \texttt{hbusreq1=0}). \tag{6}$$

As long as `hburst0` keeps changing, `START` cannot be raised, since this is forbidden by the trace in Fig. 4(b). Without `START` being raised, the bus ownership cannot change (Equation 5). The bus is initially granted to master 0 (Equation 4). When the bus remains granted to master 0 (`HMASTER=0`) forever and is requested by master 1 (`hbusreq1=1`), the fairness constraint stated in Equation 6 cannot be fulfilled.

Our heuristic is able to find a countertrace that illustrates this problem. It is presented in Fig. 4(c) together with the only output trace that conforms to the design intent and fulfills Equation 4 and 5. Equation 6 is not fulfilled.

There are multiple ways to solve this conflict. Deciding which way is best depends on the design intent and can thus not be answered automatically.

VI. Conclusions and Future Work

In this work we proposed the use of counterstrategies to debug specifications that are unrealizable or in conflict with the design intent. We also discussed techniques to keep the explanations for conflicts as simple as possible.

Countertraces give much simpler explanations than normal counterstrategies, because they are independent of the opponent's moves. We presented a heuristic algorithm without exponential blow-up of the state space that is able to compute such a countertrace for some of the cases in which one exists. Our experience and experimental results show that they are a powerful tool. To the best of our knowledge, countertraces have not been mentioned in literature before.

Minimization by means of delta debugging can greatly reduce the complexity of an unrealizable specification, while still preserving the conflict. Compared to the minimization method of Cimatti et al. [5], our technique is much faster on average. Minimized specifications allow faster computations of counterstrategies, lead to simpler games and summarizing graphs, and increase the chance of finding a countertrace. Thus, performing minimization before applying other debugging techniques is crucial for larger specifications.

In the future, we plan to evaluate our techniques on bugs that were not artificially constructed, but occurred in real specification-development processes. We already started to integrate our debugging approach into RAT [18], utilizing its graphical user interface. Our goal here is to provide high usability and interoperability with other features of RAT.

Acknowledgments: We would like to thank Amir Pnueli for suggesting the use of counterstrategies. We also thank Viktor Schuppan and anonymous reviewers for valuable comments on earlier versions.

References

[1] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime. UPPAAL-Tiga: Time for playing games! In *Proc. Computer Aided Verification (CAV'07)*, pages 121–125, 2007.

[2] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[3] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware form PSL. In *6th International Workshop on Compiler Optimization Meets Compiler Verification*, 2007. Electronic Notes in Theoretical Computer Science http://www.entcs.org/.

[4] Y. Bontemps, P.-Y. Schobbens, and C. Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamamenta Informaticae*, 62(2):139–169, 2004.

[5] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'08)*, pages 52–67, 2008.

[6] K. Claessen. A coverage analysis for safety property lists. In *Proc. Formal Methods in Computer Aided Design*, pages 139–145, 2007.

[7] S. Dellacherie. Automatic bus-protocol verification using assertions. In *GSPx*, 2004.

[8] D. Fisman, O. Kupferman, S. Seinvald, and M. Y. Vardi. A framework for inverent vacuity. In *Proc. Haifa Verification Conference (HVC)*, 2008.

[9] D. Große, U. Kühne, and R. Drechsler. Estimating functional coverage in bounded model checking. In *DATE*, pages 1176–1181, 2007.

[10] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *6th Conference on Formal Methods in Computer Aided Design (FMCAD'06)*, pages 117–124, 2006.

[11] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *Computer Aided Verification*, pages 258–262, 2007.

[12] S. Katz, O. Grumberg, and D. Geist. "Have I written enough properties?" — A method of comparison between specification and implementation. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 280–297, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[13] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[14] O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Foundations of Computer Science*, pages 531–542, Pittsburgh, PA, October 2005.

[15] M. Leucker. Model checking games for the alternation-free μ-calculus and alternating automata. In *Proc. Int. Conf. on Logic Programming and Automated Reasoning (LPAR'99)*, pages 77–91. Springer, 1999.

[16] M. Leucker and T. Noll. Truth/SLC - a parallel verification platform for concurrent systems. In *Computer Aided Verification*, pages 255–259. Springer, 2001.

[17] R. Mori and N. Yonezaki. Several Realizability Concepts in Reactive Objects. *Information Modeling and Knowledge Bases*, 1993.

[18] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal analysis of hardware requirements. In *Design Automation Conference*, pages 821–826, 2006.

[19] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[20] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[21] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[22] P. Stevens and C. Stirling. Practical model-checking using games. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1998. LNCS 1384.

[23] C. Stirling. Local model checking games. In *Proc. Concurrency Theory*, pages 1–11. Springer-Verlag, 1995.

[24] L. Tan. PlayGame: A platform for diagnostic games. In *Computer Aided Verification*, pages 492–495. Springer, 2004. LNCS 3114.

[25] S. Tripakis and K. Altisen. On-the-fly controller synthesis for discrete and dense-time systems. In *World Congress on Formal Methods*, pages 233–252, 1999.

[26] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

Connecting Pre-silicon and Post-silicon Verification

Sandip Ray
Department of Computer Sciences
University of Texas at Austin
sandip@cs.utexas.edu

Warren A. Hunt, Jr.
Department of Computer Sciences
University of Texas at Austin
hunt@cs.utexas.edu

Abstract— **We present a framework for post-silicon analysis, that provides a formal, bidirectional communication with pre-silicon verification. We show how to exploit the framework to provide a formal guarantee on post-silicon verification accuracy under limited observability. In particular, we partition a pre-silicon assertion checker (with full observability) into (1) a limited-observability checker and (2) an in-silicon integrity unit. The composition of the two units is guaranteed to provide the same accuracy as a pre-silicon checker. We apply the framework in the verification of a cache system.**

I. Introduction

Hardware verification research has traditionally focused on pre-silicon analysis. Unfortunately, the complexity of modern systems precludes catching all design errors at pre-silicon; consequently, post-silicon verification is a key component of the verification tool-flow. Post-silicon verification uses first-pass fabricated silicon to catch design errors missed at pre-silicon. Post-silicon simulations run at target clock speeds, permitting exploration of deeper design states than afforded in pre-silicon. Unfortunately, post-silicon verification is constrained by observability: only a few of the thousands of internal signals are observable during normal chip operation. The situation is exacerbated by the trend away from bus-based architectures towards point-to-point links with no central observation points [1]; many of the links exist within sockets encasing multiple processor cores and are not observable.

Post-silicon verification in industry is relatively isolated from pre-silicon. Current observability enhancement techniques entail on-chip instrumentation or hooks to monitor internal registers or bus transactions [2], [3]; however, the hooks are typically added without analysis of design invariants.

The goal of our research is to develop a unified framework connecting pre- and post-silicon analysis. We model hardware designs using a formal Hardware Description Language (HDL) deeply embedded in a formal logic. The netlist is a constant with semantics specified by a formal evaluator. This deep embedding permits analysis of functional and non-functional properties of the same design by associating different evaluators for the logical constant: our HDL has evaluators for (1) functional evaluation, (2) And/Inverter Graph, (3) gate delay, and (4) information flow [4]. We view the design artifact as a database with associated annotations, specifications, and mechanically checked theorems, providing an integrated environment for property checkers, coverage monitors, and regression and analysis routines. The database is initialized with pre-silicon results, and is interrogated during post-silicon verification. The connection between pre- and post-silicon is bidirectional: results from post-silicon augment the database, facilitating further pre-silicon analysis. To our knowledge, our work represents the first connection between pre- and post-silicon verification within a unified logical foundation.

This paper discusses one component of this framework, *e.g.*, the use of formalized assertion checkers and coverage monitors to transfer verification collateral from pre- to post-silicon. We show how to partition a pre-silicon check (with full observability) into (1) an on-chip *integrity unit*, and (2) an external unit. The integrity unit has full observability, but must have little hardware overhead. The external unit has partial observability but can assume that in-silicon analysis has succeeded. The two units are mechanically certified to provide the same guarantee as the original check. The framework provides a disciplined chip instrumentation mechanism with a formal guarantee on verification accuracy. The framework is being mechanized using the ACL2 theorem prover.

A. A Trivial Example

As an example of partitioning a property check, consider a split-transaction bus, and consider checking that the number of outstanding memory requests does not exceed a threshold. Fig. 1 shows three ways of performing the check. One approach is to snoop the bus and use external circuitry to check the count of outstanding requests (Fig. 1(A)); the observability requirement is significant, *e.g.*, all outstanding transactions. At another extreme, building the checker in silicon (Fig. 1(B)) ameliorates observability but incurs significant hardware cost. However, we can build only the up-down counter in silicon, and the comparator externally (Fig. 1(C)). The composition of the two units is provably equivalent to the original monitor; but the observability requirement is only the *count* of outstanding requests and the only hardware cost is the counter.

The above example is illustrative but pedagogical. We now list several questions that a robust partitioning mechanism must address. We will discuss our approaches to addressing them in the remainder of the paper.

- *How can we use design invariants to aid in partitioning?* Our example partition split the checker. In general, one must take into account design invariants to effectively winnow the set of observed signals.
- *How can we make the partitioning flexible to satisfy different observability requirements?* Our example exhibited

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

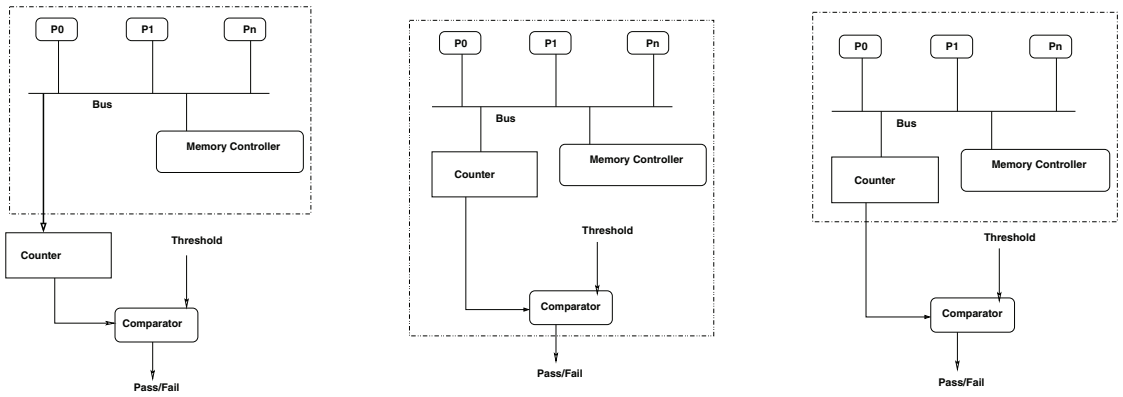


Fig. 1. Example of partitioning a post-silicon check. (A) Checker implemented externally. (B) Checker built entirely as an integrity unit in silicon. (C) Partitioned checking with a portion of checker built into silicon. In each case, the box with dotted outline contains the components built into silicon.

a single partition but in practice, different partitions might be necessary to cater to available observability.

- *How can we automate the decomposition of the monitor?* The above partition was trivial: a part of the checker circuitry was built into silicon. In practice, we need automated decomposition of monitors for complex conditions.

II. Partitioning Framework

Monitor partitioning involves (1) pre-silicon monitor $\mathcal{M}$, (2) post-silicon monitor $\mathcal{P}$, and (3) integrity unit $\mathcal{I}$.

Remark 1: In the description below, we leave implicit the set S of states and set I of inputs but assume that each state and input is represented by a tuple of Booleans. We assume the existence of a state transition function $\mathit{step} : S \times I \rightarrow S$, such that $\mathit{step}(s, i)$ returns the design state one clock cycle from s. Given a formalized HDL, $\mathit{step}(s)$ is derived by running the functional evaluator on the constant representing the design from state s. An execution from state s is a sequence of states obtained by repeated application of step for a corresponding input sequence. We restrict executions to be finite.

Pre-silicon Monitor: For our purpose, a pre-silicon monitor $\mathcal{M}$ is a function that takes a finite execution trace and returns a Boolean. We say that $\mathcal{M}$ *passes* an execution τ if it returns *true*; otherwise $\mathcal{M}$ *fails* τ. We restrict ourselves to monitors that can be defined within a decidable theory.[1] In ACL2, we use a decidable subclass of the logic called SULFA [5], which contains the basic logical constructs (*e.g.*, conjunction, disjunction, conditionals), embeds the theory of lists, and permits (bounded) recursive definitions. The evaluation of a SULFA expression reduces to a bounded check on a finite-state system which can be efficiently synthesized into a netlist. We discuss in Section III how we use this capability.

Remark 2: By the term *monitor* we include both assertion checkers and coverage monitors. In practice, checkers and monitors serve different purposes. A checker must return *true* on each execution; a failure indicates a bug. A coverage monitor determines if a specific corner case has been exercised during testing. However, the distinction is irrelevant to partitioning mechanisms.

Post-silicon Monitor: At post-silicon, only a subsequence of the execution is observable. We refer to such a trace as a *partial trace*. Formally, a partial trace σ is a subsequence of a pre-silicon execution trace τ, with guaranteed observability only of a certain pre-determined sequence of transitions. For a memory system, the guarantee might only include key communication events between the processing elements and memory controller, *e.g.*, signal transitions indicating initiation of a cache request or grants. A post-silicon monitor $\mathcal{P}$ is a function that returns a Boolean on any partial trace σ. Informally, if $\mathcal{P}$ passes a partial trace σ, then we want to guarantee the following: "σ is the subsequence of a pre-silicon trace τ such that a given (pre-silicon) monitor $\mathcal{M}$ passes τ."

Integrity Unit: It is normally impossible to guarantee post-silicon accuracy merely by analyzing a partial trace σ. Suppose σ elides a transaction that enforces a design invariant: then it is impossible to determine from σ whether the invariant is in fact enforced. The use of an integrity unit $\mathcal{I}$ circumvents this problem. $\mathcal{I}$ is a (relatively inexpensive) hardware monitor built into silicon. Given a partial trace σ, the external monitor $\mathcal{P}$ can analyze σ under the assumption that σ is a subsequence of a trace τ that passes the integrity check implemented by $\mathcal{I}$. In practice, the definition of $\mathcal{I}$ is culled from design artifacts used to enforce integrity policies: one common unit deployed in chip-sets monitors the traffic between the processor and the chip-set controller, interrupting the processor if the number of unacknowledged messages exceeds a threshold. The integrity unit does not require additional pins or I/O.

III. Post-silicon Analysis of a Memory System

We illustrate the partitioning approach on the cache coherence protocol of a multiprocessor memory system. The

[1]The decidability restriction is not germane to the framework, but imposed to facilitate the use of SAT as explained in Section III. In practice, the assertions are finite-state hardware invariants which satisfy this restriction.

protocol is loosely based on the German protocol. It uses three communication channels between processes (*clients*) and the memory controller (*home*): channel 1 carries access requests; channel 2 carries grant messages and invalidate requests; channel 3 carries invalidate acknowledgements. We implement channels by a bus-based communication interface; we also implement datapath and memory. The following communication events are relevant, and will be referred to as *critical events*.

- A client p requests a (shared or exclusive) cache block c by setting the request line for c to high in channel 1.
- Home initiates processing a request for block c from process p by setting this request line low.
- When c is granted, home signals its availability to p by setting the grant line for c to high in channel 2.
- When p receives access of c, it sets the grant line low.

The German protocol has been extensively used as a verification benchmark [6], [7], [8]. However, this paper is not about the verification of the protocol; we use it to illustrate some facets of the partitioning mechanism.[2]

Consider checking the following assertion, which is identified at pre-silicon as an invariant necessary for coherence.

Assertion: Home processes cache requests in a sequential order. That is, once Home initiates processing access request of process p for cache block c, it services no further cache access request for c until p has been granted access to c.

It is not difficult to design a pre-silicon monitor $\mathcal{M}$ for this assertion. $\mathcal{M}$ snoops the bus, tracking requests and grants between clients and the home. On the other hand, at post-silicon, observability restrictions obviously preclude recording all bus transactions. In our first simplistic scenario, we restrict our post-silicon monitor $\mathcal{P}$ to record only the critical events for a cache block c. The following theorem connects $\mathcal{P}$ with $\mathcal{M}$. The theorem has been mechanically certified by ACL2.

Theorem 1: Let $\mathcal{P}$ be a monitor that records only the critical events for cache line c. Let τ be any bounded sequence of bus transactions and σ be a subsequence of τ containing the critical events for c. If $\mathcal{P}$ passes σ then $\mathcal{M}$ passes τ.

In spite of being simplistic, $\mathcal{P}$ can uncover useful bugs; indeed, $\mathcal{P}$ was used in an early implementation to uncover the design error shown in Fig. 2. Note that the bug involves several cycles for a specific communication sequence, and is difficult to hit by random simulation (at pre- or post-silicon).

Remark 3: $\mathcal{M}$ and $\mathcal{P}$ are implemented in a decidable theory, *e.g.*, the SULFA subclass of ACL2. A consequence is that Theorem 1 is a formula in a decidable fragment of the ACL2 logic and can be discharged by SAT solving; the automation is critical to the robustness of the framework since the large number of monitors associated with a design preclude the use of theorem proving to discharge individual correctness. Furthermore, we can use SAT-based image computation on SULFA models; from the error isolated by $\mathcal{P}$ on a partial trace, image computation reconstructs a complete trace to exhibit the bug in simulation. Finally, decidability makes it possible to synthesize some functions in the definition of $\mathcal{P}$ into silicon; we return to this point after discussing integrity units.

[2]In previous work, we in fact verified a model of the protocol. That proof is not germane to this work, but it helped identify certain protocol invariants.

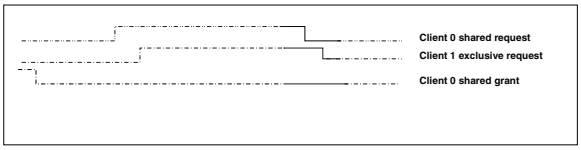


Fig. 2. A post-silicon bug uncovered by $\mathcal{P}$. Home sets request line (for shared access) from Client 0 to low before it sets the request line (for exclusive access) from Client 1 low. Yet, the request line for Client 1 is set to low before the grant line of Client 0 is set to high, violating **Assertion**. Only events represented by fall of solid edges for Clients 0 and 1 are observed. Solid lines constitute a a partial counterexample trace. Dotted lines extend this to a complete execution trace which illustrates the bug in simulation.

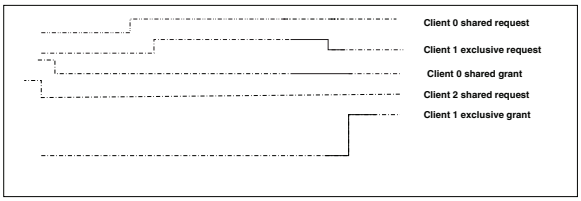


Fig. 3. A certified partially observed post-silicon trace. Only the events corresponding to the rise or fall of a solid edge are observed. Here Home initiates processing the request of Client 1 first, and does not set the access request line for any other client before granting access to Client 1. Dotted lines represent a possible extension of the trace.

Theorem 1 requires σ to contain all critical events. In practice, the number of observable events is restricted to a finite threshold. However, we reconstruct unobserved events through the integrity unit. For our assertion, the following unit $\mathcal{I}_T$ suffices. Let T be the upper bound on the number of observable bus transactions; $\mathcal{I}_T$ counts the number n of critical events among outstanding transactions, signalling an (observable) exception if n exceeds T. The following theorem (certified by ACL2) extends Theorem 1 by accounting for $\mathcal{I}_T$.

Theorem 2: Let $\mathcal{P}_T$ be a monitor that records only the critical events for a cache line c up to a threshold T. Let τ be a bounded sequence of bus transactions, σ be a subsequence of τ containing the critical events for c. If $\mathcal{I}_T$ does not signal exception on τ and $\mathcal{P}_T$ passes σ then $\mathcal{M}$ passes τ.

Theorem 2 is a statement of partial correctness, with the integrity unit providing the logical precondition. Fig. 3 shows a partial trace passed by the combination of $\mathcal{P}_T$ and $\mathcal{I}_T$. Theorem 2 guarantees that the trace indeed satisfies assertion.

The simplicity of Theorem 2 perhaps belies its flexibility. $\mathcal{P}_T$ and $\mathcal{I}_T$ merely represent a *logical* decomposition of $\mathcal{M}$. However, both are compositions of SULFA functions each of which can be mechanically synthesized into hardware description with provably equivalent semantics. Thus, when silicon real-estate is available, we can mechanically produce an augmented hardware design with some functions in the definition of $\mathcal{P}_T$ built into silicon. In the absence of available hardware, $\mathcal{I}_T$ itself can be further decomposed into in-silicon

and external components without affecting the formal guarantee: in the example, $\mathcal{I}_T$ is essentially similar to the checker in Fig. 1; thus, Fig. 1(C) represents one way of its decomposition.

The discussion above underlines the trade-off between hardware cost and logical guarantee. As the in-silicon component is augmented to include a functionality of $\mathcal{P}_T$, observability is ameliorated at the expense of hardware: building the entire $\mathcal{P}_T$ into silicon provides a complete verification accuracy at prohibitive hardware cost. In practice, the in-silicon unit is designed to provide a deterministic communication pattern for the portion of the trace not visible to the external analyzer.

IV. Related Work

One of the earliest uses of formal methods to improve post-silicon observability is by Gopalakrishnan and Chou [1]. They use constraint solving and abstract interpretation to compute state estimates for memory protocols. Ahschlager and Wilkins [9] use model checking techniques for post-silicon debug: the approach is to develop a formal property describing the observed bug and use make use of model-checking to generate a trace that encounters the bug. Safarpour *et al*. [10] use SAT solving to find circuit locations to automatically detect and repair errors using a stuck-at fault model. There has also been work on developing on-chip monitors for enhancing observability [11], [12], [13]; however, there has been little work on decomposing such monitors into on-chip and off-chip components. De Paula *et al*. [14] use SAT-solving techniques to successively "backspace" from a crashed post-silicon state to provide an execution trace that is used for off-line debugging. However, they do not address verification or provide a means for a formal guarantee on post-silicon accuracy.

V. Conclusion and Future Work

We have presented a framework for connecting pre- and post-silicon verification through formally certified partitioning of monitors. Our approach provides a flexible mechanism for transferring pre-silicon verification collateral to post-silicon, and giving a formal assurance on post-silicon accuracy. To our knowledge, our work represents the first connection between pre- and post-silicon analysis through formal proof.

One strength of our framework is the ability to reuse extant design artifacts: pre-silicon checkers and monitors, post-silicon assertions and coverage events, on-chip integrity units, etc., are available either as components of functional verification or as policy enforcement hardware. We use them judiciously to provide an integrated framework for certifying partial traces.

Our work is motivated by the desire to enable a tightly integrated, formal, bidirectional communication between pre-silicon and post-silicon. A deeply embedded HDL with formal semantics permits the interrogation of pre- and post-silicon results and annotations as database queries. Our partitioning mechanism exploits this foundation by using formal proofs as a conduit between pre- and post-silicon verification.

Our framework is in very early stages. In future work, we will tighten the connection between pre- and post-silicon verification and explore opportunities for further automation. For instance, we have only used functional verification artifacts to assist in post-silicon checking, but scalability may require cooperation from the protocol design, *e.g.*, time stamps on messages to determinize communication to assist in reconstruction of unobserved trace fragments.

Acknowledgements: This work has been funded in part by the Semiconductor Research Corporation under Grant No. 08-TJ-1849. We thank Ganesh Gopalakrishnan for insightful discussions.

References

[1] G. Gopalakrishnan and C. Chou, "The Post-silicon Verification Problem: Designing Limited Observability Checkers for Shared Memory Processors," in *5th International Workshop on Designing Correct Circuits (DCC 2004)*, M. Sheeran and T. Melham, Eds., Mar. 2004.

[2] R. Leatherman, "On-chip Instrumentation as a Verification Tool," in *FMCAD 2006 Workshop on Pre- and Post-silicon Verification: Methods and Research Opportunities*, G. Gopalakrishnan, Ed., Nov. 2006.

[3] M. Abramovici, P. Bradley, K. Dwarkanath, P. Levin, G. Memi, and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," in *Proceedings of the 43rd Design Automation Conference (DAC 2006)*. ACM/IEEE, 2006, pp. 7–12.

[4] R. S. Boyer and W. A. Hunt, Jr., "The E Language," in *International Workshop on Hardware Design and Functional Languages (HFL 2007)*, A. K. Martin, C. Seger, and M. Sheeran, Eds., 2007.

[5] E. Reeber and W. A. Hunt, Jr., "A SAT-Based Decision Procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA)," in *Proceedings of the 3rd International Joint Conference on Computer-Aided Reasoning (IJCAR 2006)*, ser. LNAI, U. Furbach and N. Shankar, Eds., vol. 4130, 2006, pp. 453–467.

[6] E. A. Emerson and V. Kahlon, "Exact and Efficient Verification of Parameterized Cache Coherence Protocols," in *Proceedings of the 12th International Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, ser. LNCS, D. Geist, Ed., vol. 2860. Springer-Verlag, July 2003, pp. 247–262.

[7] A. Pnueli, S. Ruah, and L. Zuck, "Automatic Deductive Verification with Invisible Invariants," in *Proceedings of the 7th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, ser. LNCS, T. Margaria and W. Yi, Eds., vol. 2031. Springer-Verlag, 2001, pp. 82–97.

[8] S. K. Lahiri and R. E. Bryant, "Constructing Quantified Invariants via Predicate Abstraction," in *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, ser. LNCS, B. Stefen and G. Levi, Eds., vol. 2937. Springer-Verlag, 2004, pp. 267–281.

[9] C. Ahlschlager and D. Wilkins, "Using Magellan to Diagnose Post-Silicon Bugs," *Synopsys Verification Avenue Technical Bulletin*, vol. 4, no. 3, pp. 1–5, Sept. 2004.

[10] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah, "Improved Design Debugging Using Maximum Satisfiability," in *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007)*, J. Baumgartner and M. Sheeran, Eds. Austin, TX: IEEE Computer Society, Nov. 2007, pp. 13–19.

[11] M. Boule, J. Chenard, and Z. Zilic, "Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug," in *International Conference on Computer Design*. IEEE Computer Society, 2006, pp. 294–299.

[12] A. J. Hu, J. Casas, and J. Yang, "Efficient Generation of Monitor Circuits for GSTE Assertion Graphs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD 2003)*. IEEE/ACM, Nov. 2003, pp. 154–163.

[13] S. Park and S. Mitra, "IFRA: Instruction Footprint and Recording for Post-silicon Bug Localization in Processors," in *Proceedings of the 45th Design Automation Conference (DAC 2008)*. ACM/IEEE, 2008, pp. 373–378.

[14] F. M. De Paula, M. Gort, A. J. Hu, S. Wilton, and J. Yang, "BackSpace: Formal Analysis for Post-Silicon Debug," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2008)*, A. Cimatti and R. B. Jones, Eds. Portland, OR: IEEE Computer Society, Nov. 2008, pp. 1–10.

A Verified Platform for a Gate-Level Electronic Control Unit

Sergey Tverdyshev
Saarland University, Dept. of Computer Science,
66123 Saarbrücken, Germany
Email: deru@wjpserver.cs.uni-sb.de

***Abstract*—We present the formal integration of an automotive bus controller into a formally verified gate-level computer system. This system consists of a complex processor and generic devices which run in parallel. The system specification is an instruction set architecture with concurrently running visible devices. The built system is an electronic control unit which is the base element for a distributed automotive system and its size on an FPGA is ca. 5M gate equivalents.**

I. Introduction

The result of this paper is a part of Verisoft [1] project which targets formal and pervasive verification of computer systems. Pervasive verification [1], [2] attempts to verify systems completely including the interaction of all components, thus minimizing the number of system assumptions.

The goal of the subproject Verisoft-Automotive is to verify an automatic emergency call system, eCall [3]. The eCall system is based on a time triggered distributed real-time system which consists of distributed hardware and a distributed operating system. The hardware part consists of several electronic control units (ECU) which can communicate with each other via a FlexRay-like bus [4]. Every ECU consists of a processor and an automotive bus controller [5], [6].

The overview of the system stack is presented on Fig. 1. The stack starts at the gate-level (hardware) which is abstracted as an assembly-level model. On top of the assembly model resides the communicating virtual machines model (CVM) [7], [8] which is implemented in a subset of C language with assembly code. On top of the stack there is a time-triggered operating system which hosts user applications, e.g. eCall.

In this paper we present the implementation and formal verification of a hardware platform for an ECU. This platform is a complex gate-level computer system, which consists of a pipelined processor and a number of memory mapped I/O devices. The gate-level system is verified against an instruction set architecture model as seen by an assembly programmer (cf. Fig. 1). This model consists of a processor specification (describing the effects of instruction execution) and specifications of the devices. While at the gate-level the processor and the devices run in parallel, at the architecture level they run concurrently, i.e. perform execution steps non-deterministically one after each other.

This work was supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

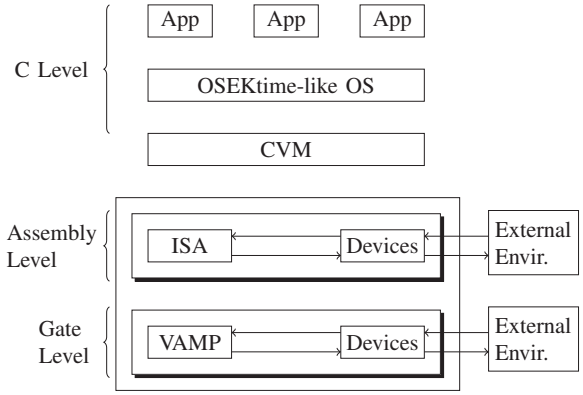


Fig. 1. Overview of the automotive computer system stack.

We build ECU by integrating an automotive bus controller into verified computer system. We also synthesise the built ECU and run it on a Xilinx FPGA.

A. Context and Related Work

The hardware computer system described in this paper forms the basis of a verified system stack developed and verified in the Verisoft project [1], [9]. The hardware core is based on the VAMP processor [10], [11], which is, to the best of our knowledge, the most complex verified processor in the open literature. The VAMP processor was first verified in the interactive theorem prover PVS. Because the rest of the Verisoft systems stack and eCall system are developed in Isabelle/HOL we developed and verified the VAMP processor again in Isabelle/HOL to guarantee the pervasiveness of our verification effort. While we reused the old top-level proof strategy, we simultaneously succeeded in speeding up the verification by solving lower-level proof goals with our automatic hardware verification environment IHaVeIt integrated into Isabelle [12]. We could decrease user interaction by ca. 40% compared to the purely interactive proofs in PVS [13].

The closest related work in systems verification is the famous CLI stack [14]. This stack consists of a non-pipelined processor, an assembler, a compiler for a simple high-level language, and an elementary operating system kernel, but is lacking external devices. In 2002 J S. Moore [2], the principal researcher of the CLI stack project, declared the

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

formal verification of a computer system with devices a grand challenge.

Devices are often modeled and verified as stand-alone systems [15], [16]. For instance, a device is modeled at some level and then certain model properties are checked. To use devices in a hardware / software system, though, they have to be modeled at different levels, as we do here.

In Verisoft-Automotive one deals with the verification of a time triggered bus interface device, which is used in the distributed automotive systems. The verification of such a system requires considering this interface at different levels, e.g. low-level for clock synchronisation in serial interface [17], gate-level implementation, and real-time properties of the software driver for this bus interface. These models and proofs are finally combined into one pervasive correctness proof. A *paper-and-pencil* style description for the latter work can be found in Knapp and Paul [5]. In this paper we show how a verified electronic control unit (ECU) for the distributed system can be constructed. This ECU is an instantiation of the gate-level computer system which is presented in this work.

Hillebrand et al. [18] previously presented the *paper-and-pencil* formalisations of a system with a hard disk drive. They define the system at the gate and the assembly level and sketch correctness arguments relating these models. However, their arguments are complicated and depend on the concrete device model. By changing the sampling of external interrupt in the implementation we obtain a generic and much cleaner (and formally verified) solution (cf. Sect. III-A3).

Alkassar et al. [19] present a concurrent assembly-level model for a processor with I/O memory mapped devices and some paper-and-pencil formalizations. The model is used in [20] to formally prove the correctness of a hard disk driver at the assembly level. Moreover, in the Verisoft project the assembly model has been connected to the machine-code model presented here [9].

II. Assembly-Level Computer System

We define the instruction set architecture (ISA) of a processor with a number of memory-mapped devices D_i (cf. Fig. 2). Compared to regular ISA definitions in addition to the processor state, the combined architecture also includes device states. Processor and devices may interact (i) by the processor accessing devices and (ii) by the device causing interrupts. Devices can also make steps on their own when interacting with an external environment (e.g. a network). Therefore we model the computation of ISA with devices as a concurrent computation.

A. Processor

A processor configuration c_{P} is a tuple consisting of (i) two program counters $c_{\mathrm{P}}.pc$ and $c_{\mathrm{P}}.dpc$ implementing delayed branching, (ii) general purpose, floating point, and special purpose register files $c_{\mathrm{P}}.gpr$, $c_{\mathrm{P}}.fpr$, $c_{\mathrm{P}}.spr$, and (iii) a byte addressable memory $c_{\mathrm{P}}.m$.

Devices are mapped into the processor memory and can be accessed by the processor via regular read and write

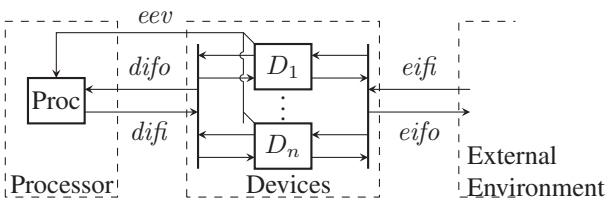


Fig. 2. Overview of a computer system with processor and devices

operations. In addition devices can signal an interrupt to the processor via an external event signal (cf. Fig. 2).

Let DA denote the set of memory addresses mapping to devices, which are disjoint from regular physical memory addresses. The processor indicates an access to a device via the device interface input $difi$ and receives the device's response on the device interface output $difo$.

Formally, let the predicates $lw(c_{\mathrm{P}})$ and $sw(c_{\mathrm{P}})$ indicate load and store word instructions and let $ea(c_{\mathrm{P}})$ and $RD(c_{\mathrm{P}})$ denote the address and affected processor register for such operations (see Müller and Paul [21] for full definitions).

The device interface input has the following four components: (i) the read flag $difi.rd = lw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a load from a device address, (ii) the write flag $difi.wr = sw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a store to a device address, (iii) the address $difi.a = ea(c_{\mathrm{P}})$ is set to the effective address, with $ea[14:12]$ specifying the accessed device and $ea[11:2]$ specifying the accessed device port (we support up to eight devices with up to 1024 ports of width 32 bits), and finally (iv) the data input $difi.din = c_{\mathrm{P}}.gpr[RD(c_{\mathrm{P}})]$ is set to the store operand.

The device interface output $difo \in \{0,1\}^{32}$ contains the device's response for a load operation on a device.

The processor model is defined by the output function Ω_{P} and the next state function Δ_{P}. The former takes a processor state c_{P} and computes a device input $difi$ to the device as defined above. The latter takes a processor state c_{P}, a device output $difo$, and an external event vector eev (where $eev[i]$ is set iff device D_i signals an interrupt). It returns the next processor state c'_{P}.

The configurations of all devices are represented as a mapping c_{D} from device indices i to the corresponding device configuration.

Our device model is sequential in the sense that a device may progress either due to a processor access or an input from the external environment. To distinguish both cases we extend the set of device indices by the processor index P and denote this set by PD.

The device transition function Δ_{D} specifies the interaction of the devices with the processor and the external environment. It takes a processor-device index $idx \in PD$, an input from the external environment $eifi$, an input from the processor $difi$, and a combined device configuration c_{D}. It returns a new device configuration $c_{\mathrm{D}}{}'$, an output to the processor $difo$, and an external output $eifo$.

Depending on the input index idx and the device input

difi, the transition function Δ_{D} is defined according to the following three cases: (i) If $idx \neq P$, the device idx makes step with the external input *eifi* and the given *difi* is ignored. (ii) If $idx = P \wedge (difi.wr \vee difi.rd)$, a device step is triggered by a processor access. In this case Δ_{D} ignores the given *eifi* and produces an arbitrary *eifo*. The accessed device and the access-type is specified by the given *difi*. (iii) Otherwise the processor does not access any device and Δ_{D} does nothing. The device output function Ω_{D} computes the external event vector *eev* for the processor based on the current device configurations.

B. Combined System

By combining the processor and device models we obtain a model for the overall system with devices as depicted in Fig. 2. This model allows interaction with environment via *eifi* and *eifo* whereas the communication between processor and devices is not visible from the outside anymore.

A configuration of the combined model consists of a processor configuration $c.c_{\mathrm{P}}$ and device configurations $c.c_{\mathrm{D}}$. We define a transition function Δ_{PD} and an output function Ω_{PD}. Both functions take the same three inputs: a processor-device index idx, a combined configuration $c.(c_{\mathrm{P}}, c_{\mathrm{D}})$, and an external input *eifi*.

We introduce some more notation for the transition and the output function. Let $difi = \Omega_{\mathrm{P}}(c.c_{\mathrm{P}})$ be the input from the processor to the devices. Let *eifi* be input from external environment. Let $(c'.c_{\mathrm{D}}, difo, eifo) = \Delta_{\mathrm{D}}(idx, c.c_{\mathrm{D}}, eifi, difi)$ denote the updated device configuration, the device output to the processor, and the external output. Let $eev = \Omega_{\mathrm{D}}(c'.c_{\mathrm{D}})$ denote the external event vector, which is computed based on the updated device configuration. Finally, if $idx = P$ then $c'.c_{\mathrm{P}}$ denotes the updated processor configuration, i.e. $c'.c_{\mathrm{P}} = \Delta_{\mathrm{P}}(c.c_{\mathrm{P}}, eev, difo)$. Otherwise $c'.c_{\mathrm{P}}$ denotes the unchanged processor configuration, i.e. $c'.c_{\mathrm{P}} = c.c_{\mathrm{P}}$.

The transition function Δ_{PD} returns the new configuration $c'.(c_{\mathrm{P}}, c_{\mathrm{D}})$. The output function Ω_{PD} simply returns the output to the external environment *eifo*.

A run of the combined system is defined for a given number of steps and a computational sequence $\sigma \in \mathbb{N} \to PD$. The latter designates the interleaving of the processor and device steps. A run starts with an initial configuration c^0. During a run the system receives inputs from the external environment as an input sequence $c.eifi \in \mathbb{N} \to eifi$ that maps step numbers to the corresponding inputs. A run for i steps results in a new configuration $c^{(i,\sigma)}.(c_{\mathrm{P}}, c_{\mathrm{D}})$ and external output sequence $c.eifo^{(i,\sigma)} \in \mathbb{N} \to eifo$. The new configuration is computed by recursive application of Δ_{PD} and the output sequence by application of Ω_{PD}.

C. Usage of Assembly-Level Computer System

The introduced assembly-level computer system provides a programming and verification model for assembly programs. For example, let us consider a run of an operating system on a computer system with a processor P, a serial interface SI, a hard disk drive HDD, and a keyboard Kbd (Fig. 3). Let us also assume that this run contains execution of a driver for HDD device. Usually, such a driver is an assembly program. At the gate level, execution of this driver can last over thousands of cycles with system components running in parallel (Sec. III-A). Obviously, we do not want to prove any *program* properties at this level. Moreover, while proving properties of this assembly driver, we do not want to consider those processor and device steps which are irrelevant to the driver execution. The presented assembly-level system provides a suitable gate-level abstraction. For example, Alkassar [22] shows that this assembly-level system allows reordering of processor and device steps if they are not depend on each other. Let us assume for the running example that SI and Kbd do not interact with the processor, and hence, those steps can be reordered (Fig. 3) and be grouped into one continuous subsequence. Now, we can prove any specific properties solely a about hard disk driver ignoring other devices. Moreover, we can treat this subsequence as an atomic driver step and provide the OS programmer with the correctness/properties of the whole driver execution. Alkassar et al [23] presented how this approach can be applied for a paging mechanism, which is implemented in a page fault handler.

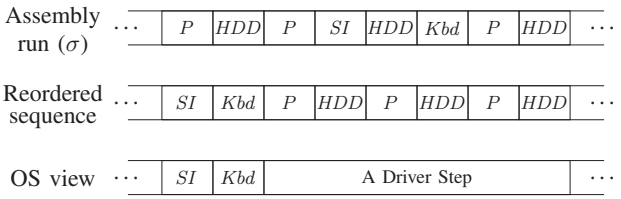


Fig. 3. Reordering and abstracting of computational sequences.

III. Gate-Level Computer System

In this section we define a computer system at the gate level. This system has a similar structure to the system introduce above, i.e. it consists of the processor VAMP and a generic model for the implementation of external devices. The processor and the devices are connected via a common bus. All components of the system run in parallel and are clocked with the same hardware clock.

A. Processor

The VAMP processor [24] is a pipelined processor with out-of-order execution. Figure 4 gives an overview of the data path of the VAMP processor. The pipeline of the VAMP consists of five stages. In stage IF an instruction is fetched from the memory. In stage ID the instruction is decoded and put in a reservation station. The source operands are read from the processor's registers or forwarding paths. Both stages operate in order, realizing a pipelined implementation of the delayed-PC architecture [21] with one delay slot per control-flow instruction. The stages EX, C, and WB implement the Tomasulo algorithm [25] for out-of-order execution. In stage EX the instruction is executed, i.e. the result of the instruction is computed. In this stage memory and device accesses, if needed, are executed. The instruction result is then put on

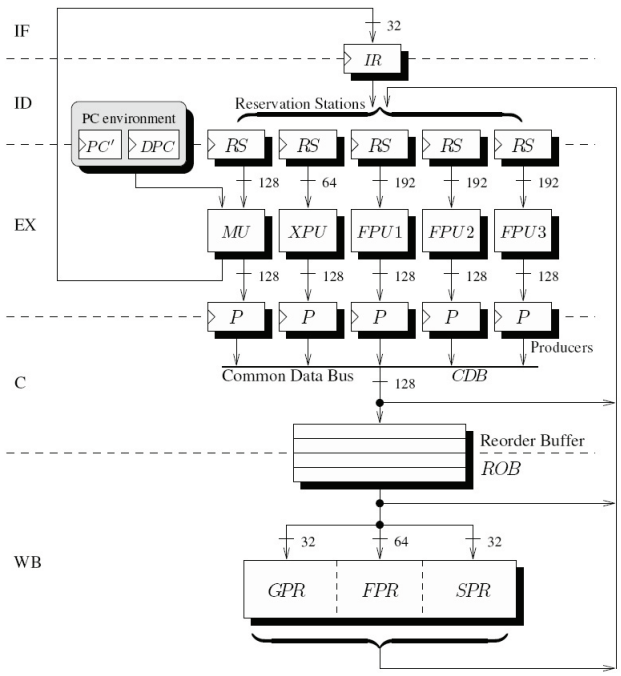


Fig. 4. Data path of the VAMP

the common data bus in stage C. Finally, in stage WB the computed result is written back to the register file in-order. This is implemented via a reorder buffer (ROB). As soon as the oldest instruction result in the ROB becomes valid, it can be written back to the register file. The write back is the last step of the instruction execution and it signals that the instruction is leaving the processor. The ROB is also used to implement precise interrupts. If an instruction is interrupted, previous instructions are written back and later instructions are flushed. The interrupted instruction is written back or flushed depending on the interrupt type (repeat or continue). We distinguish two interrupt sources: internal (e.g. for page fault or overflow) and external (e.g. for reset or I/O interrupts). Internal interrupts are computed during instruction execution while external interrupts are an additional input bus of the VAMP.

For a detailed description of the processor core implementation see [26].

1) The VAMP Communication Interfaces: The VAMP communicates with two "off-the-chip" components: the external devices and the memory. The processor employs the following communication buses: (i) device interface input and output ($difi$ and $difo$), (ii) memory interface input and output ($mifi$ and $mifo$), and (iii) an external event vector eev with which devices signal their interrupts.

The VAMP accesses devices by setting the bit $difi.req$. The type of access (read or write) is given by bit $difi.w$. The accessed address is set on $difi.a \in \mathbb{B}^{30}$ and data for the write access are on $difi.din \in \mathbb{B}^{32}$. The accessed device sets $difo.reqp$ to signal that it is processing a request and cannot accept a new one. An active value of bit $difo.brdy$ signals

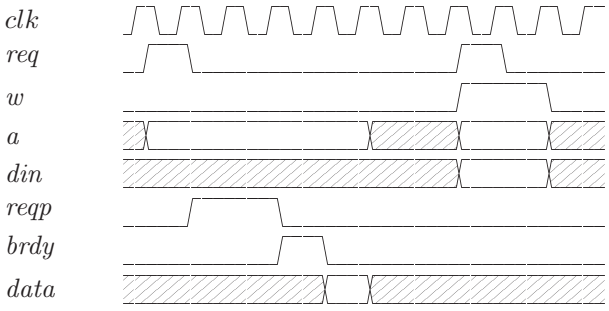


Fig. 5. Timing of device read and write accesses

that in the next cycle the request is finished and the data on $difo.dout \in \mathbb{B}^{32}$ will be valid. Figure 5 shows a typical processor-device communication. The memory protocol is similar.

2) Memory: Memory is a non-modeled external component. We define its content by observing memory interfaces. Let M^0 be an initial memory content. Let $mifi^t$ and $mifo^t$ be the state of the memory buses at cycle t. We introduce a function $bw(mifi^t, M^{t-1}(a)) \in \mathbb{B}^{64}$ which computes the new content of the cell on address a. Let predicate $busy^t$ signal the end of a memory access. Let the shorthand $wr^t \triangleq mifi^t.bwb \neq 0^8$ denote if a non zero number of bytes is written. The memory content M^t at cycle $t > 0$ is recursively defined as $M^t(a) = bw(mifi^t, M^{t-1}(a))$ if $mifi^t.a = a \wedge \neg busy^t \wedge wr^t$ and $M^t(a) = M^{t-1}(a)$ else.

3) Sampling External Interrupts: Previously, the VAMP has sampled external interrupts on the eev bus in the WB stage [24]. This is problematic, however, when relating the gate-level implementation to the instruction-set architecture. For example, consider an instruction that clears (acknowledges) a device interrupt. In the implementation, after the device access completes the device lowers its interrupt and the instruction leaves the memory unit. To reach the WB stage, the instruction needs several hardware cycles (cf. Fig. 4). In this time frame the device may reactivate its interrupt, which will then be sampled for the same instruction in the WB stage. From an assembly programmer's point of view, however, the new interrupt belongs to, or should affect, the next instruction.

In [18] an (informal) device-specific solution was proposed for this problem, still sampling all external interrupts in the WB stage. Here, we solve this problem generically and hence much more elegantly. Revisiting the instruction-set architecture, we may distinguish two types of device accesses: active ones that are performed when executing load / store instruction on device addresses and passive reads that occur when external interrupts are sampled. This interpretation reveals that *every* instruction (with external interrupts enabled) executes at least one device access and that active accesses are always accompanied by a passive read. To avoid any "shadow" scenarios devices should be accessed exactly once per instruction. Obviously, the instructions without any active

device access already satisfy this requirement. For the other instructions we satisfy the requirement by sampling externals interrupts at the time when the device access completes. Thus, the access result and the external interrupts are read at the same cycle.

Notably, other implementations described in literature do not solve the above problem. For example, in [27] and in the MIPS-R3000 family [28, Chapter 8] external interrupt are sampled before instruction issue and before the memory stage, respectively. Both implementations still may access external devices twice.

B. Devices

All gate-level devices make a step in every cycle, which is the main difference to the ISA-level devices. Configurations of all devices are combined in a mapping h_{D} from device indices i to the corresponding device configuration. An input from external environment to all devices is represented as a mapping $eifis$ from device indices to device inputs. Similarly, an output from all devices is a mapping $eifos$ from device indices to device outputs.

The step function δ_{D} defines the behavior of devices. It takes input from the environment for all devices $eifis$, processor input $difi$, state of all devices h_{D}, and $reset$ bit. It produces new state h'_{D}, outputs to the processor $difo$, output to the environment $eifos$, and external interrupts for the processor eev.

We make an assumption that δ_{D} obey the processor-device protocol (Fig. 5).

C. Combined System

The gate-level computer system consists of the VAMP and the generic device model. The VAMP and devices communicate via the internal device interface buses $difi$, $difo$, and eev. These buses introduce one cycle delay in the communication, e.g. the VAMP places the request on the $difi$ and at the next cycle the device reads these data.

The next-state function updates the processor and the devices by the application of their next-state functions. It also defines the communication between the system components and the external world. For the combined VAMP-Devices system the external world consists of the memory chips and the external environment of the devices. Formally, the step function computes the next state $h'_{\mathrm{PD}}.(h_{\mathrm{P}}, h_{\mathrm{D}})$ based on a given system state $h_{\mathrm{PD}}.(h_{\mathrm{P}}, h_{\mathrm{D}})$, inputs from the device environment $eifis$ and the memory $mifo$. It also computes the outputs to the external environment $eifos$ of the devices and to the memory $mifi$.

A run of the combined model is defined recursively by the application of the next-state function for a given number of hardware cycles. We employ $h^t.eifis$ to denote the device inputs from the external environment at cycle t. We denote the input from the external memory at cycle t as $h^t.mifo$. A run for t cycles starting from a configuration h^{init}_{PD} results in a new configuration $h^t.(h_{\mathrm{P}}, h_{\mathrm{D}})$ outputs to the external environment $h^t.eifos$, and outputs to the external memory $h^t.mifi$.

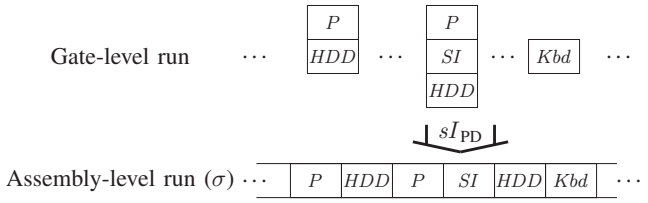


Fig. 6. Abstraction of hardware cycles to a computational sequence

IV. Computer System Correctness Criterion

We define a scheduling function which relates gate-level runs with their ISA-level counterparts. Its definition is based on special hardware events indicating progress at the gate level, which has to be reflected at the ISA-level. We split events into two groups, processor-sided and external-environment-sided.

1) Processor-sided events: The result of any instruction without memory or device access is finally computed at the write back stage, i.e. at the cycle when the instruction is leaving the VAMP. We use the notation wb^t to denote the value of the hardware write-back signal at cycle t.

An instruction writing to the memory or accessing a device forces *irreversible* changes of the memory / device when the access ends. The access cycle precedes the write back cycle of the instruction. Thus, external state is altered before the instruction leaves the processor. Therefore, memory and device correctness have to be treated specially. To built a precise relation, we need a cycle where the result of a device access is put on the bus (Fig. 5). We denote this event at cycle t by da^t. Note that da has almost the same semantics as wb, because once a device access is done its effect cannot be rolled back.

2) External-environment-sided events: Device steps can be triggered by the external environment. We introduce a function $DevIds$ which, for a given device state h_{D} and the external inputs $eifis$, computes a computational sequence $\sigma = DevIds(h_{\mathrm{D}}, eifis)$ consisting of only device identifiers. Thus, sequence σ corresponds to the devices making step due to that input $eifis$. The definition of this function depends on the concrete device instances and how the gate-level device model has to be abstracted. The only restriction we put on $DevIds$ is that it should not return a sequence with duplicate device identifiers. We present two examples for function $DevIds$: (i) modeling every implementation step in the specification, i.e. $DevIds$ always returns a computational sequence which contains all device identifiers. (ii) purging those steps of a gate-level device which have no effect, e.g. if the state of the device did not change, its identifier is not in the result of $DevIds$.

We define the scheduling function sI_{PD}. For a given number of hardware cycles it returns a computational sequence σ. Figure 6 depicts how a gate-level run for a system with three devices can be mapped to a computational sequence. Function sI_{PD} is defined by recursion on hardware cycles. Let us abbreviate $\sigma^t = sI_{\mathrm{PD}}(t)$. At cycle zero we return the empty sequence $[\,]$, i.e. $\sigma^0 = [\,]$. For the recursion step

let boolean flag was_da^t be true if there was the end of a device access but this instruction is not yet written back, i.e. $was\_da^t = \exists t' < t\,.\, da^{t'} \wedge \forall t'' \in]t' : t[\,.\, \neg wb^{t''}$. We define the recursive step of sI_{PD} as follows:

$$\sigma^{t+1} = \sigma^t \circ \begin{cases} DevIds(h^t.h_{\mathrm{D}}, h^t.eifis) \circ [P] & \text{if } da^t \\ [P] \circ DevIds(h^t.h_{\mathrm{D}}, h^t.eifis) & \text{if } wb^t \wedge \neg da^t \\ & \quad \wedge \neg was\_da \\ DevIds(h^t.h_{\mathrm{D}}, h^t.eifis) & \text{otherwise} \end{cases}$$

A. Assumptions

We can only state and prove a relation between the gate-level and the ISA-level models if they receive the same inputs. We define a predicate $sync\_eifis$ that tests if inputs from the external environments between cycles ts and te are synchronised. If a device with identifier $\sigma^{te}(j)$ makes a step at $t \mapsto t+1$ (with $ts \leq t < te$), this device makes this step with the input $h^t.eifis(idx)$. Then, we compare input for the device $h^t.eifis(idx)$ with the specification input $c.eifi^j$ for step j. Similarly, we define a predicate $sync\_eifos$ that tests whether an output sequence to the external environment of implementation and specification match.

The gate-level and the ISA-level device models are based on the generic step functions δ_{D} and Δ_{D}, respectively. We keep them generic to instantiate them with needed devices. To state anything about runs of these models, we have to specify how results of these functions are related with each other.

We introduce a predicate sim_D that holds if the implementation states of all devices are in a relation with their specification. We make a "one-step" assumption specifying a relationship between one step of the-gate level device model and a sequence of steps in the specification of devices with respect to sI_{PD}. We define this assumption for *any* initial state and *any* input sequence.

$$\begin{aligned} &sim_D(h^0.h_{\mathrm{D}}, c^{\sigma^0}.c_{\mathrm{D}}) \ \wedge sync\_eifis(0,1,h.eifis,c.eifi) \ \wedge \\ &h.difi = c^{\sigma^1}.difi \implies \\ &\quad sim_D(h^1.h_{\mathrm{D}}, c^{\sigma^1}.c_{\mathrm{D}}) \ \wedge \\ &\quad sync\_eifos(0,1,h.eifos, c^{\sigma^1}.eifo) \ \wedge \\ &\quad h^1.difo.dout = c^{\sigma^1}.difo \ \wedge h^1.eev = c^{\sigma^1}.eev \end{aligned}$$

B. Software Conditions

Sometimes it is impossible or too expensive to implement handling of some exceptions in the hardware. These special cases restrict the software which can be executed on the developed hardware. We call these restrictions *software conditions* [10]. We present two software conditions for the VAMP processor, which also hold for the gate-level computer system.

The first condition excludes RAW hazards for the self-modifying assembly code [24], [29]. This condition makes use of so-called *sync*-instructions, which drain the processor pipeline. Other than that a sync-instruction should act as a no-op. In the VAMP the instruction *movs2i IEEEf R0* has sync semantics. Hillebrand [30] observed that jumping to and returning from an interrupt service routine has sync semantics, too. The software condition requires at least one sync instruction between any two instructions which produce a RAW hazard for instruction fetch.

In the presence of external devices, we need another software condition which guarantees the absence of accesses to the undefined address space. The main issue is the liveness because neither memory nor devices respond to an access to the undefined address space, and hence, such an access never terminates.

C. The Simulation Theorem

Our correctness criterion states that every implementation run can be simulated by a specification run. The time notions, hardware cycles and computational sequences, are related via a function sI_{PD}. Device states and system outputs are related via sim_D and $sync\_eifos$, respectively. We introduce a predicate $Rconf(h_{\mathrm{P}}, c_{\mathrm{P}})$ which tests processor states.

The VAMP has more registers than the ISA processor. The registers, which are present in both models, are called the *visible registers* (from a programmer's point of view). These include general purpose registers and program counters. Other implementation registers are called *invisible*. They hold partial results of instruction execution, e.g. the internal registers of the functional units. The predicate $Rconf(h_{\mathrm{P}}, c_{\mathrm{P}})$ tests the visible registers of the VAMP and the ISA.

Some VAMP components, e.g. the program counters, may have values that never occur in the specification. Therefore, *the* correctness criterion of the VAMP is stated after every interrupt [24], which is signaled by signal $JISR$. At these cycles all visible registers of VAMP and ISA must have equal values.

We can now formulate the simulation theorem. Let inputs for the processor-device models be synchronised with respect to $sync\_eifis$ and let the initial states be equivalent. Let the assembly code satisfy the software conditions. Let $\sigma^t = sI_{\mathrm{PD}}(t)$. We show that the gate-level model after t cycles and ISA-level model after executing σ^t have equivalent states and produce equal outputs.

$$\begin{aligned} &Rconf(h^0.h_{\mathrm{P}}, c^{[]}.c.c_{\mathrm{P}}) \wedge M^0 = c^{[]}.c.c_{\mathrm{P}}.M \wedge \\ &sim_D(h^0.h_{\mathrm{D}}, c^{\sigma^0}.c.c_{\mathrm{D}}) \wedge sync\_eifis(0,t,h.eifis,c.eifi) \implies \\ &\quad Rconf(h^t.h_{\mathrm{P}}, c^{\sigma^t}.c.c_{\mathrm{P}}) \wedge M^t = c^{\sigma^t}.c.c_{\mathrm{P}}.M \wedge \\ &\quad sim_D(h^t.h_{\mathrm{D}}, c^{\sigma^t}.c.c_{\mathrm{D}}) \ \wedge \\ &\quad sync\_eifos(0,t,h.eifos, c^{\sigma^t}.eifo) \end{aligned}$$

The theorem proof can be found in [13]. Note that in the proof of this theorem the shadow scenarios for sampling interrupts (Sect. III-A3) show up.

V. Electronic Control Unit

In Verisoft-Automotive project [1] an automotive system is built, which is placed and run on a vehicle. The system consists of a time-triggered bus system (inspired by FlexRay [4]) and a time-triggered operating system (inspired by OSEKTime [31]). On top of these system components, an emergency-call application eCall [3] is realized.

This automotive system at the gate-level consist of several electronic control units (ECU) which are connected via

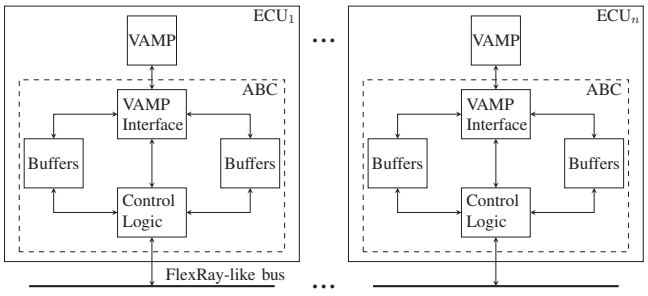


Fig. 7. Automotive system with several ECUs

FlexRay-like bus [4] (Fig. 7). A *paper-and-pencil* correctness proof of the whole system is presented in [5], [6].

In this section we present a single ECU which consists of the VAMP processor and a bus interface. The bus interface is called the automotive bus controller or the ABC device.

The device is placed between a processor and a time-triggered bus, and it can be architecturally split into two parts. One part is only visible from the bus side and another from the processor side. The main components of every part are a send buffer and a receive buffer. Since the buffers on the bus and the processor side are independent, the device can service both sides simultaneously. The buffers on the bus side are used to hold the outgoing messages to the bus (the send buffer) and to store the incoming messages from the bus (the receive buffer). The buffers on the processor side are memory mapped into the VAMP address space. There are special hardware cycles when the roles of the buffers of the processor and the bus sides are swapped, i.e. the buffers of the processor side become the buffers of the bus side and vise versa. This implements the data transfer between the processor and the bus sides of the ABC device.

Correctness of a single ECU cannot be treated fully decoupled from the rest of the automotive system. During a run of the automotive system, the ECUs exchange data via a time-triggered bus. Such a run is split into so called time slots. One of the main slot characteristics is the absence of buffer swappings. The buffers are swapped at the border of two slots. For example, the processor of an ECU can only read/write data during a slot. Therefore, the correctness of an ECU is split into two parts: a local one and a distributed one. The local correctness captures the communication of the processor and the ABC device during one slot. The distributed correctness states that during a run the exchange of the data between ECUs is correct, and that this run can be decomposed into time slots. In this work we are interested in only local correctness of ECU.

In order to construct a verified ECU we have to instantiate the introduced computer systems with the ABC device. The whole instantiation consists of several simple steps: (i) Instantiate the step function of the implementation of the device model, i.e. use implementation of the ABC device to define δ_{D}, (ii) Instantiate the step function of the specification of the device model, i.e. use specification of the ABC device to define Δ_{D}, (iii) Instantiate the trigger functions da and $DevIds$, (iv) Instantiate the simulation relation sim_D, (v) Prove that the definition of da and $DevIds$ satisfies their intended semantics, (vi) Prove the assumptions which guarantee that the device implementation can be simulated by the device specification.

The first four steps instantiate the generic devices. The last two steps guarantee that the instantiated models preserves its properties. We have done all steps but the last one which is work in progress.

By instantiating the generic device theory, we created two models of the ECU. One is the ECU model on the gate level: this is the gate-level model with the VAMP and the ABC device. Another is the ECU model at the assembly level with the ISA-processor and the specification of the ABC device. The correctness criterion of the gate-level ECU is easily derived from the simulation theorem presented above. Note, for this paper we only had to assume the sixth step of the instantiation for the ECU because ABC correctness has to be provided by the device developers and it's on-going work.

VI. Summary

We have presented a formally verified computer system at the gate-level, consisting of a processor and external devices. We are the first to present a running formally verified ECU which extends our previous work [32].

The base of the computer system is a pipelined processor, which is called VAMP. It is a 32-bit RISC processor featuring out-of-order execution with five functional units, precise interrupts, and address translation. In contrast to previous work, we verified the VAMP in the context of pervasive system verification, considering also external devices.

We have introduced generic device models for gate and ISA levels and combined them with the corresponding processor models. Thus, have built and formally verified a computer system with generic memory mapped devices. The formal verification of the combined system, allowed us to establish a clean semantics of the external interrupts.

We have shown that external devices can be easily integrated into the computer system. We integrated a controller for a time-triggered bus and the resulting system can be used as an electronic control unit (ECU) in a distributed automotive system. These results are formalized and mechanically verified in the interactive theorem prover Isabelle/HOL. We also synthesised and ran the created ECU on an FPGA and the unit size is ca. 5M gate equivalents (this is joint work with Andrey Shadrin).

Table I presents verification efforts in the theorem prover Isabelle/HOL in terms of person years, number of theorems, and number of proof steps. We also used hardware verification environment IHaVeIt [12] to reduce user's involvement in the proof process.

A. Future Work

We see several interesting topics for future work. Finishing formal pervasive verification of the whole eCall system. In this paper we presented the formally verified platform for an

TABLE I
VERIFICATION EFFORTS IN ISABELLE/HOL WITH IHAVEIT

Prover	Target	P.y.	Theorems	Proof steps
Isabelle	VAMP (no FPU, MU)	1.5	1206	20455
	Devices	0.5	52	967
	Combining systems	0.7	118	2714
	Total	2.7	1376	24316

electronic control unit but we didn't verify the automotive bus controller device. There is an on-going work about formal verification of a gate-level implementation of ABC device which includes: low-level bus modelling, clock synchronisation and scheduling algorithm for several ECUs. Another work in progress at our chair is verification of a distributed operating system which runs on top of the gate-level system with several ECUs [33]. The last step in this topic will be formally putting all parts together.

REFERENCES

[1] The Verisoft Consortium, "The Verisoft Project," http://www.verisoft.de/, 2003.

[2] J. S. Moore, "A grand challenge proposal for formal methods: A verified stack." in *10th Anniversary Colloquium of UNU/IIST*, ser. LNCS, B. K. Aichernig and T. S. E. Maibaum, Eds., vol. 2757. Springer, 2002, pp. 161–172.

[3] European Commission (DG Enterprise andDGInformation Society), "eSafety forum: Summary report 2003. Technical report, eSafety," March 2003.

[4] F. Consortium, "FlexRay – the communication system for advanced automotive control applications," http://www.flexray.com/, 2006.

[5] S. Knapp and W. Paul, "Pervasive verification of distributed real-time systems," in *Software System Reliability and Security*, ser. IOS Press, NATO Security Through Science Series., M. Broy, J. Grünbauer, and T. Hoare, Eds., vol. 9, 2007.

[6] S. Knapp, "Pervasive layered verification of a distributed real-time system," in *Third International Conference on Systems (ICONS'08)*, 2008, pp. 323–328.

[7] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul, "On the correctness of operating system kernels," in *TPHOLs 2005*, ser. LNCS, J. Hurd and T. Melham, Eds. Springer, 2005. [Online]. Available: http://busserver.cs.uni-sb.de/publikationen/GHLP05.pdf

[8] T. In der Rieden and A. Tsyban, "CVM – a verified framework for microkernel programmers," in *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B.V., 2008.

[9] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban, "Balancing the load: Leveraging semantics stack for systems verification," *JAR: Special Issue on Operating Systems Verification*, 2009, to appear.

[10] I. Dalinger, M. Hillebrand, and W. Paul, "On the verification of memory management mechanisms," in *CHARME 2005*, ser. LNCS, D. Borrione and W. Paul, Eds. Springer, 2005, pp. 301–316.

[11] S. Beyer, C. Jacobi, D. Kroening, D. Leinenbach, and W. Paul, "Putting it all together: Formal verification of the VAMP," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4–5, pp. 411–430, Aug. 2006.

[12] S. Tverdyshev and E. Alkassar, "Efficient bit-level model reductions for automated hardware verification," in *TIME 2008*. IEEE, 2008, pp. 164–172.

[13] S. Tverdyshev, "Formal verification of gate-level computer systems," Ph.D. dissertation, Saarland University, Saarbrücken, May 2009.

[14] W. R. Bevier, W. A. Hunt, S. Moore, and W. D. Young, "An approach to systems verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 411–428, 1989.

[15] G. Berry, M. Kishinevsky, and S. Singh, "System level design and verification using a synchronous language," in *ICCAD*, 2003, pp. 433–440.

[16] ALDEC – The Design Verification Company, "UART nVS," http://www.aldec.com/products/ipcores/_datasheets/nSys/UART_nVS.pdf, 2006.

[17] J. Schmaltz, "A formal model of lower system layer," in *FMCAD'06*. IEEE/ACM Press, 2006, pp. 191–192.

[18] M. Hillebrand, T. In der Rieden, and W. Paul, "Dealing with I/O devices in the context of pervasive system verification," in *ICCD 2005*. IEEE, 2005, pp. 309–316.

[19] E. Alkassar, M. Hillebrand, S. Knapp, R. Rusev, and S. Tverdyshev, "Formal device and programming model for a serial interface," in *VERIFY 2007*, ser. CEUR Workshop Proceedings, B. Beckert, Ed., vol. 259. CEUR-WS.org, 2007, pp. 4–20. [Online]. Available: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/paper04.pdf

[20] E. Alkassar and M. A. Hillebrand, "Formal functional verification of device drivers," in *VSTTE 2008*, ser. LNCS, N. Shankar and J. Woodcock, Eds., vol. 5295. Springer, 2008, pp. 225–239.

[21] S. M. Mueller and W. J. Paul, *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[22] E. Alkassar, "Os verication extended - on the formal verication of device drivers and the correctness of client server software (draft)," Ph.D. dissertation, Saarland University, Saarbrücken, 2009.

[23] E. Alkassar, N. Schirmer, and A. Starostin, "Formal pervasive verification of a paging mechanism," in *14th intl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), to appear*, ser. LNCS. Springer, 2008.

[24] S. Beyer, "Putting it all together: Formal verification of the VAMP," Ph.D. dissertation, Saarland University, Saarbrücken, 2005.

[25] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[26] D. Kröning, "Formal verification of pipelined microprocessors," Ph.D. dissertation, Saarland University, Saarbrücken, 2001.

[27] J. E. Smith and A. R. Pleszkun, "Implementing precise interrupts in pipelined processors," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 562–573, 1988.

[28] R.-J. Brüss, *RISC Die MIPS-R3000-Familie*, ser. Architektur, Systembausteine, Compiler, Tools, Anwendungen. Siemens, 1991.

[29] I. Dalinger, "Formal verification of a processor with memory management units," Ph.D. dissertation, Saarland University, Saarbrüucken, 2006.

[30] M. Hillebrand, "Address spaces and virtual memory: Specification, implementation, and correctness," Ph.D. dissertation, Saarland University, Saarbrücken, 2005.

[31] O. Consortium, "OSEK VDX portal," http://portal.osek-vdx.org/, 1993.

[32] M. Hillebrand and S. Tverdyshev, "Formal verification of gate-level computer systems (to appear)," in *4th International Computer Science Symposium in Russia*, ser. LNCS, A. R. A. Morozov, K. Wagner and A. Frid., Eds., vol. 5675. Springer, 2009, pp. 322–333.

[33] M. Schmidt, "Verification of distributed operating systems (draft)," Ph.D. dissertation, Saarland University, Saarbrücken, 2009.

Protocol verification using flows: An industrial experience

John O'Leary
Intel
john.w.oleary@intel.com

Murali Talupur
Intel
murali.talupur@intel.com

Mark R. Tuttle
Intel
tuttle@acm.org

***Abstract*—We prove the parameterized correctness of one of the largest cache coherence protocols being used in modern multi-core processors today. Our approach is a generalization of a method we described last year that uses data type reduction and compositional reasoning to iteratively abstract and refine the protocol and uses invariants derived from protocol "flows" to make the abstraction-refinement loop converge. Our prior work demonstrated the value of sequencing information that appeared within the linear flows describing a protocol in design documents. This paper extends the notion of flows to capture intricate scenarios seen in real industrial protocols and demonstrates that there is also valuable information in the interaction among flows. We further show that judicious use of flows is required to make the method converge and identify which flows are most suitable.**

I. Introduction

We validated an extremely complex cache coherence protocol that will soon appear in multi-core processors from Intel. We used a generalization of the method we reported last year [1] based on the CMP method [2], [3], [4] augmented with message flows. This protocol, which we call LCP, is a high-performance protocol that is designed to be scalable to a large number of cores. Such intricate distributed protocols are especially susceptible to functional bugs that standard techniques like testing and simulation are unlikely to find and consequently formal verification is indispensable in their validation. We think LCP may be one of the largest, most complicated cache coherence protocols ever validated with formal methods. As one measure, the Flash cache coherence protocol, to which only a handful of formal methods have been successfully applied, has about 10 Boolean state variables per process and 16 different message types in all. In contrast, with over 70 Boolean state variables per process and around 50 message types, the state space for LCP is several orders of magnitude larger than Flash (see Section II).

While many techniques [5], [6], [7], [8], [9] have been proposed for parametric protocol verification, none of them scale well to large protocols, and those that do scale [10], [11] require an inordinate amount of manual effort to succeed. The CMP method [1], [2], [3], [4] is the only method for parametric verification we are aware of that scales to large protocols and is easy to use. It is an interactive proof method based on compositional reasoning that uses a model checker as a proof assistant. Though it combines the best of theorem proving and model checking, the main difficulty in applying this method is coming up the *non-interference lemmas* or invariants to guide the proof. As in theorem proving, this is a time-consuming process requiring a thorough knowledge of the protocol. Moreover, adding one wrong invariant can lead the proof astray and render subsequent work useless.

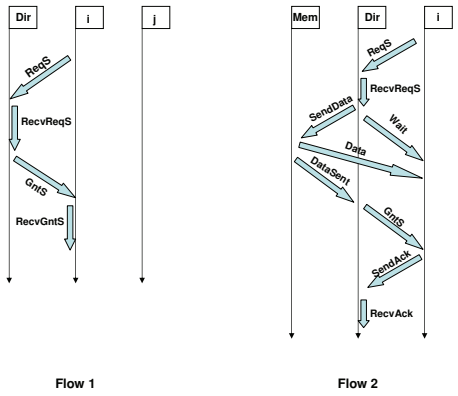


Fig. 1. Message flows as a linear sequence or acyclic graph of events.

In our earlier paper [1], we showed that the burden of generating the non-interference lemmas required by CMP can be significantly reduced by using the message *flows* typically found in industrial design documents. Flows are linear sequences of system events such as sending and receiving messages in the case of distributed message-passing systems, as illustrated on the left of Figure 1. We demonstrated the efficacy of our method by applying it to academic protocols, namely, German's protocol and the Flash protocol.

In this paper, we describe a generalization of the method presented in [1] and its application to the LCP cache coherence protocol. The primary contributions of this paper are:

1) Generalizing flows from linear traces to directed acyclic graphs, like the flow on the right of Figure 1, and a simple language for describing flows.
2) Demonstrating that we can derive powerful non-interference lemmas from constraints on events occurring in different flows, and not just constraints on events occurring within a single flow. Simply stating that two flows cannot be in progress at the same time, for example, can dramatically speed up the convergence of the CMP method.
3) Demonstrating that not all flows are equally useful, and

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

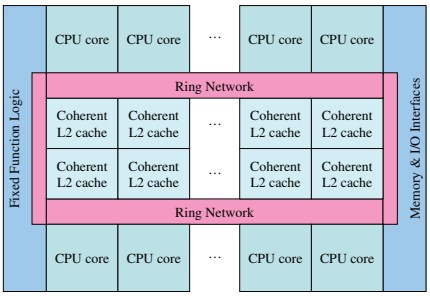


Fig. 2. Schematic of the Larrabee multi-core architecture

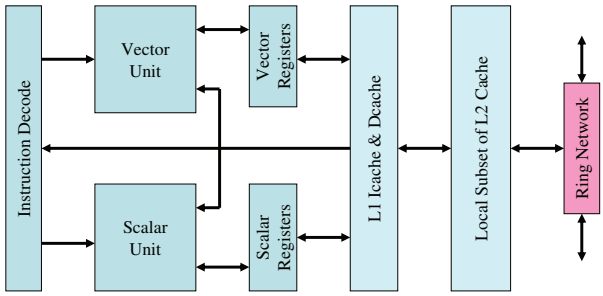


Fig. 3. Larrabee CPU core and associated system blocks

that a more judicious use of the information in flows can also speed the convergence of the CMP method.

4) Parametrically verifying the correctness of the Intel cache coherence protocol LCP for any number of processors.

In verifying LCP we used a total of 15 flows, all easily obtained from the design documents, to derive around 36 lemmas. To make the CMP method converge another 5 lemmas had to be supplied by hand. A similar effort earlier [12] where we verified a cache protocol of comparable size using the CMP method required us to supply nearly 25 lemmas manually. Clearly, flows lead to a dramatic reduction in the number manually supplied lemmas and makes it much easier to use the CMP method.

The rest of the paper is structured as follows. In the next section we describe the salient features of the LCP protocol. In Section III we discuss the possible alternatives to the CMP method and why they are inadequate. An overview of the CMP plus flows method of [1] is given in Section IV followed by a discussion of the extensions required to deal with LCP in the next section. In Section VI, we present a new language to capture richer flows and also show how to derive stronger constraints than just simple precedence constraints. A detailed description of our experience using these extended flows to verify the LCP protocol is given in Section VII. Section VIII concludes the paper.

II. Larrabee and LCP

Larrabee is the code name for a multi-core visual computing architecture under development at Intel Corporation [13]. The Larrabee architecture is based on a set of CPU cores that run the x86 instruction set, extended with support for vector processing operations and some specialized scalar instructions. Figure 2 shows a schematic of the architecture. Each core is associated with its own subset of a coherent L2 cache that affords fast, high-bandwidth data access to each core and simplifies data sharing and synchronization. The number of CPU cores and the number and type of co-processors and I/O blocks are implementation dependent, as are the positions of CPU and non-CPU blocks on the chip. To validate the LCP protocol in full generality we need parametric reasoning.

Figure 3 shows the major functional blocks in a single core. Larrabee's global second-level (L2) cache is divided into separate subsets, one for each CPU core. Each CPU has direct access to its own subset of the L2 cache. Data read by a CPU core is stored in its L2 cache subset and can be accessed quickly, in parallel with other CPUs accessing their own local L2 cache subsets. Data written by a CPU core is stored in its own L2 cache subset and is flushed from other subsets, if necessary. Larrabee uses a bi-directional ring network to allow agents such as CPU cores, L2 caches and other logic blocks to communicate with each other within the chip. The LCP (Larrabee coherence protocol) runs on the ring network and maintains coherency of shared data.

As shown in Figure 4, our model of the Larrabee coherence protocol is organized as a parameterized number of identical caching agents which talk to a central directory that controls access to the data items. For the purpose of verifying the coherence protocol our model abstracts away the ring structure and assumes point-to-point communication links between the agents (including links between the caches and from the off-chip memory to individual caches).

Unlike the Flash protocol where the directory distinguishes between local requests and external requests, the LCP makes no such distinction. This means when verifying two index properties it is enough to retain two cache agents concretely in the abstract model whereas for Flash we had to keep three agents, one local agent and two non-local agents [1]. In addition to these agents, there is also a memory controller that talks to the directory and supplies memory lines that have not yet been imported onto the chip.

The high-level model we verified preserved much of the internal structure of each caching agent. Thus, apart from the L2 cache we also had the L1 cache and actions of the agent depended on the states of both the caches. Further, the various in- and out-message buffers and related bookkeeping data structures were also modeled. Other than assuming point-to-point links between the various agents, we modeled almost every significant detail of the protocol which increased the complexity of flows/transactions considerably.

The complexity of a protocol can judged by the number of different types of messages that are exchanged by the agents. German's protocol for example has only 7 different messages, the Flash protocol, considered hard to verify, has 16 different types and LCP has around 50 different message types (comparable to the protocol we verified in [12]). In terms

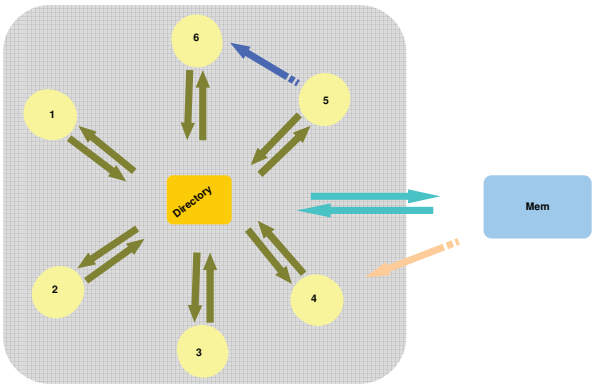


Fig. 4. Basic Organization of the LCP model

of Boolean variables, each process in LCP has approximately 70 variables. In contrast the Flash protocol has around 10 state variables per process. The Murphi description of the LCP protocol actions was about 3000 lines whereas Flash has around 1000 lines.

III. Protocol Verification Techniques

Broadly, there are two classes of techniques to verify distributed protocols: model checking methods that aim for maximum automation and theorem proving methods that aim for scalability.

A. Model Checking Methods

Several techniques like Indexed Predicates [6], [14], Counter Abstraction [5] and related methods [15], Regular Model Checking [16], [17], the Split Invariants method [18] and Environment Abstraction [19], [20] have been proposed to deal with verification of distributed protocols. While these methods have a higher degree of automation than the one we present, their scalability remains to be proven.

1) Indexed Predicates Method: Indexed predicate abstraction generalizes predicate abstraction to predicates that have free index variables. Given a protocol $P(N)$ and collection $\mathcal{I}$ of indexed predicates this method automatically produces the strongest universally quantified invariant of $P(N)$ over $\mathcal{I}$. Intricate systems like the Bakery protocol and Tomasulo's out-of-order processor have been verified by using indexed predicates [6], but scalability remains an issue. Even for as small an algorithm as German's protocol the indexed predicates method takes several hours to produce an invariant.

2) Cutoffs based approach: Another approach to verifying a parameterized system $P(N)$ is to find a cutoff k such that verifying $P(k)$ is enough to guarantee the correctness of $P(N)$ for any value of N. There has been some work on this topic [7], [21] and related topics [22], [23] but the cutoffs are large making them impractical to use. For example, in [21] a cutoff of 7 was found for a directory based protocol. But real protocols are so large that even verifying a system with 3 agents is often not possible.

3) Counter and Environment Abstractions: Counter Abstraction [5] and its generalization Environment Abstraction [19] are based on the idea of partitioning a collection of identical agents into equivalence classes based on the predicates they satisfy and for each partition tracking only one representative. The abstract models produced by these methods tend to be very detailed and consequently too large as we look at bigger protocols. Environment abstraction, for example, is just barely able to handle a simplified version of the Flash protocol [24].

B. Theorem Proving Methods

Apart from classical theorem proving there are methods like aggregated transactions [11] and disjunctive invariants [25] that are user-guided. These suffer from the well-known problem of having to provide guidance in minute detail to get the proof through. For example, the disjunctive invariants method requires the user to supply a set of additional predicates whose disjunction is inductive. It is unlikely that theorem proving style methods can be used practically to verify large protocols given that just to verify the Flash protocol the aggregated transactions method took a couple of days of effort.

C. The CMP Method

The CMP method straddles the categories of model checking and theorem proving based methods: it uses a model checker as a proof assistant to carry out user guided proof. The crucial advantage of the CMP method is that the user supplied lemmas don't have to comprise an inductive invariant [1]. This means the amount of guidance required is a lot less than in theorem proving methods. Using the CMP method we have earlier verified the Flash protocol [1] and another large cache coherence protocol within Intel [12]. The latter protocol is comparable in size to the LCP protocol. In our experience, CMP method is the only viable method currently for handling large protocols and our effort was to make it more usable by reducing the lemma burden as much as possible.

IV. Description of the CMP Method

For the rest of the paper we will use the same system model as in [1]. In particular, we consider a symmetric protocol P with N processors $[1..N]$ whose transition relation is given as a collection of rules. Each rule is a *guarded command* written as

$$rl : \forall i, j.\rho \rightarrow a \qquad \text{or} \qquad rl(i,j) : \rho(i,j) \rightarrow a(i,j)$$

where rl is the *rule name*, ρ is an expression called the *guard*, a is a list of assignments called the the *action* and i, j are the process index variables.

The CMP method is a compositional reasoning based method and it consists of two basic steps — *abstraction* and *strengthening* — that are applied iteratively to a protocol as shown in Figure 5.

Given a property $I = \forall i, j \in [1..N].I(i,j)$ that we want to prove is an invariant of P, the CMP method creates an

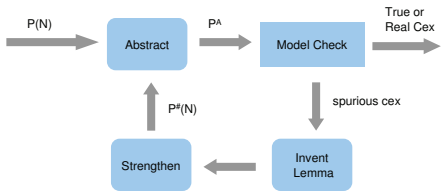


Fig. 5. The CMP method

abstract model $\widehat{P}$ that retains two agents, say $1, 2$ without loss of generality, and replaces the rest of the processes with a highly non-deterministic process $Other$. Intuitively, in a symmetric system if a property $\phi(1, 2)$ holds for processes $1, 2$ then $\phi(i, j)$ holds for any other pair of processes i, j as well. Thus, it is enough to consider an abstract model with detailed information on $1, 2$ and check if $I(1, 2)$ holds. The abstraction process in CMP method is inexpensive as it is almost syntactic and the bulk of the state space of $\widehat{P}$ comes from processes 1 and 2. On the flip side $\widehat{P}$ tends to be very coarse as the behavior of $Other$ is completely unconstrained. To get rid of the resulting spurious counterexamples, the CMP method requires the user to provide *non-interference lemmas* or invariants that are used to refine the abstract model. This process is continued iteratively till we find a real counterexample or prove that $I(1, 2)$ holds. Pseudo-code for the method is given below.

CMP(P,I) =
  $P^{\#} = P$; $I^{\#} = I$
  while $abstract(P^{\#}) \not\models abstract(I^{\#}(1, 2))$ do
    examine counterexample *cex*
    exit if *cex* is a real counterexample
    find $L = \forall i, j.L(i, j)$ ruling out *cex*
    $P^{\#} = strengthen(P^{\#}, L)$
    $I^{\#}(i, j) = I^{\#}(i, j) \wedge L(i, j)$
  end

If the loop terminates normally, the method and protocol symmetry allow us to conclude that $I^{\#}$ and consequently I are invariants of P. If the loop terminates via the exit, then either I or one of the proposed lemmas L is not an invariant of the protocol, and the user must back up and try again.

In McMillan's work [2] the abstraction operation used was data type reduction [26] which essentially throws away all the state variables of processes $[3..N]$. Our analysis of the CMP method in [1] allows us to use richer abstractions than data type reduction. In particular our method allows meaningful abstraction of the auxiliary variables used to track flows.

A. Making CMP better using Flows

Not surprisingly the main difficulty in applying the CMP method is coming up with the non-interference lemmas. As demonstrated in [1], message flows or simply flows yield powerful invariants that can be used as non-interference lemmas in the CMP method and reduce the number of lemmas we need to supply by hand.

The flows used in [1] are linear orderings of events usually involving two agents, see Figure 1. Each entry in a flow is either a simple event, corresponding to a single rule firing in the protocol, or a *sub-flow* where a sub-flow is itself a flow composed of simple events. The notion of sub-flow serves to chop up a complicated flow into smaller units such that each unit shows interaction between two agents.

The constraints derived from flows are the implicit precedence constraints between events occurring in the flows. For example, according to the first flow in Figure 1, $ReqS$ has to happen before $RecvReqS$ can happen. This can be converted into a precise lemma by having a set $Aux(i)$ of auxiliary variables for each process i that track all the flows i is involved in and for each flow the last rule that was fired by i. In case of $RecvReqS$ action the precedence constraint is simply that if, for process i, the $RecvReqS$ action is enabled then there must be an auxiliary variable recording the fact that i was involved in $ReqS$ action earlier. These simple precedence constraints turn out to be surprisingly powerful as invariants.

The advantage of flows is that they are intuitive to understand and readily available in the design documents. Flows in fact allow us to state the core ideas that go into the design of a protocol in terms of higher level concepts while avoiding the specific implementation details. This means flows are quite robust and resistant to changes in the protocol design which makes them very attractive as user supplied annotations.

V. Extensions to Flows

Apart from the local caching agents and the directory, real cache protocols have other types of agents, like the off-die memory controller, which add to the complexity of the interaction between the agents. Flows designed to capture two agent interactions are no longer sufficient.

Consider Flow 2 of Figure 1 which calls for an extension to our notion of flows. Here a process i requests access to an item that has not yet been fetched onto the chip. The on-chip directory forwards the request to the off-chip memory controller along with the id of the requesting agent. The directory also sends an acknowledgment to the caching agent. The memory controller for its part sends the required item to the caching agent and also sends a message completion to the directory. On receiving the message completion the directory sends a grant message to the agent. On receiving both the grant message from directory and the data message from the memory controller agent i transitions to shared state and sends a completion message to the directory. The transaction ends when the directory receives the completion message.

This scenario is similar to, and typical of, the complex interactions present in LCP and it differs from the flow shown in the left of Figure 1 in crucial ways. The interaction between the three agents is tightly coupled and it is not possible to identify meaningful sub-flows that have only two agents involved. In [1], even though we had more than two agents

involved, it was to easy see that each flow was made up of logical sub-units consisting of only two agents.

Further, an event might have multiple preceding actions. For instance, event *SendAck* can happen only after the agent has received both the data and grant messages. Receiving only one of these is not sufficient to enable the *SendAck* message. Linear flows cannot capture multiple dependency.

Similarly, an event in the flow might have more than one succeeding event. For instance, the $RecvReqS$ event leads to two further events $SendData$ and $Wait$. With linear flows the number of events depending on a given event is at most one.

Finally, unlike Flow 1 of Figure 1 which totally orders its events, Flow 2 imposes only a partial order. For instance, there is no ordering between the events $Data$ and $GntS$; they can be received by i in any order. One way to deal with this is to flatten the partial order into a collection of total orders. But this runs into two issues: first, the number of resulting flows might be large and unnecessarily obscure the simplicity present in the dag representation. Second, having more flows also means we will have to introduce more auxiliary variables or extend their ranges and thus, we will also end up making the augmented model bigger.

Thus, it is clear that a new notion of flows is needed that is expressive enough to capture directed acyclic graphs (dags). Moreover, note that in Flow 2 of Figure 1 there are three primary agents in the flow but that each event involves at most two agents. So apart from specifying events we also have to specify which agents are involved in the events. A new flow language that takes into account these extensions is described in the next section.

VI. Generalized Flows

To keep the presentation simple we omit treatment of subflows in this section since they are not required for LCP.

A. Language for Flows

A flow is denoted by

$$(\mathit{flow}, \mathit{conflicts}) : \{\mathit{prec}_1, \ldots, \mathit{prec}_n\}$$

where *flow* is the name of the flow, *conflicts* is a set of flows, and each $prec_i$ is a *precedence* of the form

$$\mathit{event} : \{\mathit{event}_1, \ldots, (\mathit{event}_m\}$$

where each $event$ is a triple $(rule, id, agents)$ specifying a particular firing of rule $rule$ by processes in the list $agents$. The meaning of a precedence is that each rule $rule_i$ instantiated with the list of agents $agents_i$ must occur in an execution before the rule $rule$ instantiated with the list of agents $agents$ can occur. We say each $(rule_i, id_i, agents_i)$ *precedes* $(rule, id, agents)$ in the flow. For example, in the flow on the right side of Figure 1, the first precedence is

$$(ReqS, id_1, \langle Dir, i\rangle) : \{\}$$

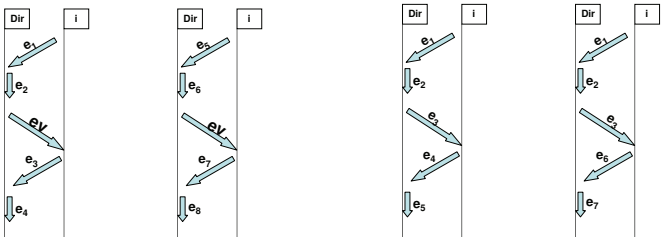


Fig. 6. Flows can share a common event (left) and a common prefix (right).

and the third precedence is

$$(SendData, id_2, \langle Mem, Dir\rangle) : \{(RecvReqS, id_3, \langle Dir\rangle)\}$$

The meaning of a conflict set is that a flow f and a flow f' in the conflict set of f cannot be alive at the same time. A flow is said to be *alive* if some rule in the flow has occurred and one of its successors has not occurred. The conflict set of f might contain f itself. In this case the meaning is that there can be only one instantiation of f alive in the system. The constraints resulting from these conflict sets turn out to be very powerful in constraining the $Other$ process.

Finally, we associate an id with each event since a particular instantiation of a rule may occur in multiple flows. The left side of Figure 6 illustrates that we must distinguish the occurrences of ev in the flows f_1 and f_2, so that the occurrence of ev in f_1 will enable the event e_3 in f_1 and not e_7 in f_2. The right side of Figure 6 illustrates that we cannot use the flow name as the event id, since different flows can share a common prefix. As long as the system is in the prefix, we don't know whether the system is in flow f_1 or f_2. In fact, because we don't know which flow the system is in, an event in the prefix must have the same event id in both flows.

B. Tracking Flows

We use auxiliary variables to track active flows as in [1]. We will assume that a pair (rl, id) appears only once in a flow.

Define $last(rl, id, p)$ in a flow f to be the set of pairs (rl', id') for which there exist agent lists ag and ag' both containing p such that (rl', id', ag') precedes (rl, id, ag) in f. We require that $last(rl, id, p)$ be the same for all flows containing (rl, id). Intuitively, this means the prefix for (rl, id) must be the same in all flows in which it appears.

Define $next(rl, id, p)$ in a flow f to be the set of pairs (rl', id') for which there exist agent lists ag and ag' both containing p such that (rl, id, ag) precedes (rl', id', ag') in f.

Define the *out degree* of (rl, id) for p in f to be the size of $next(rl, id, p)$ in f. We require that the out degree of (rl, id) for p be the same in all flows containing (rl, id).

Define $flows(rl, id)$ to be the set of flows f containing an event (rl, id, ag) for some agent list ag. This is the set of all flows containing (rl, id) and by definition one of them is active when an agent executes (rl, id).

For each process p, we maintain a set $Aux(p)$ of tuples of the form (rl, id, od, fls) where (rl, id) is an instance of

$update(p, rl(i,j), id)$ =
  for each $(rl', id') \in last(rl, id, p)$
    aux = choose $(rl', id', od, fl\_set) \in Aux(p)$
      such that $flows(rl, id) \cap fl\_set \neq \{\}$
    new_aux = if $od > 1$
      then $\{(rl', id', od - 1, fl\_set \cap flows(rl, id))\}$
      else $\{\}$;
    $Aux(p) := Aux(p) \setminus \{aux\} \cup new\_aux$
  if (rl, id) precedes no other rule then
    return $Aux(p)$ else
    return $Aux(p) \cup (rl, id, outdegree(rl, id), flows(rl, id))$

Fig. 7. Tracking flows with auxiliary variables. This procedure describes how p updates $Aux(p)$ when an event (rl, id, ag) involving p is executed. The choose operator throws an error if there is nothing to choose.

a rule rl that has fired with an instantiation including p, od is the number of successors of (rl, id) in a flow that have yet to fire,[1] and fls is the set of candidates for the flow — containing (rl, id) — currently in progress. When (rl, id) fires, we add this tuple to the set with od initialized to the out degree of (rl, id) for p and fls to the set $flows(rl, id)$ of all flows containing (rl, id). As each successor (rl', id') to (rl, id) in a flow fires, we decrement od by 1 and restrict the set fls to those flows containing (rl', id') in addition to (rl, id). When od drops to 0, we remove the tuple from the set. This procedure is given by the *update* procedure in Figure 7.

One unanswered question about the *update* procedure is how we choose the value of id when an instantiation $rl(p_1, p_2)$ of a rule $rl(i, j)$ fires.[2] The answer is that we try every value for id such that (rl, id) appears in a flow. Most values of id will fail because the choose operator will fail to find a satisfactory tuple in the auxiliary set of tuples and throw an exception. We use any value of id that succeeds (and undo the effects of prior attempts that failed). For LCP, corresponding to each rule rl there was only one id that appeared in the flows, so updating auxiliary variables was simpler than the general procedure given here.

C. Lemmas from Flows

We derive two classes of lemmas from flows. The first class is an extension of the "precedence" lemmas derived in [1] to the richer flows. The second class uses the conflict sets.

1) Precedence Constraints: We define the precondition for firing the rule rl with the processes i and j by

$$pre_{i,j}(rl) = \bigvee_{id} pre_{i,j}(rl, id).$$

We define the precondition for firing (rl, id) with i and j by

$$pre_{i,j}(rl, id) = pre_i(rl, id) \wedge pre_j(rl, id).$$

We define the precondition for firing (rl, id) with i by

$$pre_i(rl, id) = \bigwedge_{(rl', id') \in last(rl, id, i)} pre_i(rl, id)(rl', id')$$

[1]Alternatively, we could maintain the actual set of successors yet to fire.
[2]A rule involving more or less than two agents is handled similarly.

where $last(rl, id, i)$ is the set of instances (rl', id') involving i that precede (rl, id) in a flow, and similarly for j. We define the precondition for firing (rl, id) with i given the prior firing of (rl', id') by

$$\begin{aligned} pre_i(rl, id)(rl', id') = \\ \exists\, od', fls'. \quad & (rl', id', od', fls') \in Aux(i) \\ & \wedge\, od' > 0 \\ & \wedge\, fls' \cap flows(rl, id) \neq \{\} \end{aligned}$$

If the guard for the rule $rl(i, j)$ is $\rho(i, j)$, then the precedence non-interference lemma is

$$\rho(i, j) \Rightarrow pre_{i,j}(rl).$$

2) Conflict Constraints: If two flows f^1 and f^2 conflict, then both cannot be active at the same time, meaning that while f^1 is active f^2 cannot start. Let

$$(rl^1_1, id^1_1, \_) \ldots (rl^1_m, id^1_m, \_)$$

be the set of events appearing in f^1 and

$$(rl^2_1, id^2_1, \_) \ldots (rl^2_n, id^2_n, \_)$$

be set of *initial* events appearing in f^2, meaning that these events have no preceding events in f^2. If f^1 is active, then some process p must have fired some rule rl^1_k in f^1, so some tuple of the form $(rl^1_k, id^1_k, \_, \_)$ must appear in $Aux(p)$. The flow f^2 cannot start if for every rule rl^2_ℓ starting f^2 and for every pair of processes i and j that might fire rl^2_ℓ, the guard $\rho^2_\ell(i, j)$ of $rl^2_\ell(i, j)$ is false. Our first conflict non-interference lemma is

$$\begin{aligned} \exists p \;\; \exists (rl^1_k, id^1_k, \_).\;\; & (rl^1_k, id^1_k, \_, \_) \in Aux(p) \\ & \Rightarrow \forall (rl^2_\ell, id^2_\ell, \_)\;\; \forall i, j.\;\; \neg \rho^2_\ell(i, j) \end{aligned}$$

where $\rho^2_\ell(i, j)$ is the guard of the rule $rl^2_\ell(i, j)$.

If the flow f^1 is in conflict with itself, then a process firing a rule in the flow cannot fire that rule again until the flow has ended, since the flow conflicts with itself. Our second conflict non-interference lemma is

$$\exists (rl^1_k, id^1_k, \_, \_) \in Aux(i) \Rightarrow \forall j.\;\; \neg \rho^1_k(i, j).$$

The two conflict lemmas are similar, but the first prevents firing of initial rules and the second prevents refiring of any rule. The second lemma turned out to be surprisingly helpful in restricting the $Other$ process in the abstract model.

VII. Verifying the LCP Protocol

In this section we describe our experience applying the CMP method in conjunction with flow based lemmas to verify the LCP protocol.

A. Obtaining the Flows

The flows we considered during verification were all readily available in the design documents written by the architects. In fact, the scenarios listed in the design documents had more information than we needed. Intuitively, only those parts of the flows that involve the directory (in other words the place where all the coordination happens) yield useful invariants.

Apart from identifying the useful parts of the flows, we had to annotate the events with the agents involved and also identify the conflict set for each flow. Both these steps were straightforward.

B. Protocol Model

The Murphi model that we verified was quite hierarchical with each rule having an extremely large body covering a variety of cases (as it was semi-automatically generated from tables describing the protocol). For instance, there is only one rule specifying the behavior of a cache agent in case it receives an invalidate message. The guard of the rule only checks the type of the incoming message and the body of the rule considers the various sub-cases depending on the state of the L1/L2 caches and the data structures. The behavior of the cache in each of the sub-cases might be quite different with differing messages being sent out.

In other words, the events associated with each of the sub-cases are quite different though they all belong to the same Murphi rule syntactically. This is not conducive to our flow based methodology which depends on being able to track events precisely. To allow precise tracking of flows, we broke up large rules into smaller ones so that each rule corresponded to a specific event mentioned in the flows. This was accomplished by simply lifting some of the branch conditions in the body of a rule to the guard.[3] After this step each event mentioned in the flows mapped to a Murphi rule.

C. Proof details

The properties that we proved were the standard safety property requiring that if there is a cache with exclusive access to a data item then no other cache has access to that item and properties specifying consistency between the directory's list of caches with access to a data item and the real access each local cache has.

To carry out the abstraction and refinement with lemmas we used a tool called *Abster* written in Ocaml. Abster takes in a parameterized protocol written in Murphi and creates an intermediate abstract syntax tree (AST) for the protocol. All the abstraction and refinement operations take place on the AST. We specify the number of agents, usually 2 or 3, to be retained concretely in the abstract model depending on the protocol to be verified. The flows are specified in a separate file. Abster automatically constructs flow lemmas and adds them to the guards of the appropriate rules (a rule whose guard appears in the flow invariant gets modified by that invariant).

[3]This had to be done only for some of the rules. Similarly, not all branch conditions had to be lifted.

To carry out the proof we used 15 flows from design documents. These led to 36 flow lemmas, with 25 of them coming from precedence constraints and the rest from conflict constraints. At first we used only the precedence constraints and proceeded to refine the abstract model with hand supplied lemmas. But we soon realized that several counterexamples were caused by two conflicting flows starting at the same time. More precisely, suppose one of the concrete agents, say 1, is involved in a *Request Shared* transaction. Since the $Other$ process is not fully constrained, it might start sending conflicting messages corresponding to *Request Exclusive* transaction to the directory. One way to prevent this it to write lemmas by hand and refine the $Other$ process. The simpler option is to use the conflict constraints that prevent a *Request Exclusive* transaction from starting while a *Request Shared* transaction is alive. Together with precedence constraints this ensures that no rule in a conflicting flow can fire.

Another cause of spurious counterexamples was the $Other$ process repeatedly firing the same rule from a flow. Suppose (rl', id') precedes (rl, id) and some other event. That is, the out degree of (rl', id') is 2. Since we are only tracking the number of successors of (rl', id') and not the precise names, (rl, id) can get fired twice by the $Other$ process. This does not happen for the concrete agents because they have state variables controlling their execution which is not the case with $Other$ process. Adding conflict constraints fixes this problem as well.

To complete the proof it was necessary to add 5 lemmas by hand. Since we had in- and out-message buffers, characterizing when a cache had access to a given item was hard. A grant shared message might be sitting in the in-message buffer, for instance, though the L2 state may not reflect it. We used the CMP method (without the flow invariants) earlier to verify a protocol of similar size though less intricate because of simpler internal structure of each cache [12]. There we had to add 25 lemmas by hand. Compared to that effort the reduction obtained using flows is dramatic and clearly makes the CMP method much more usable. This experience once again confirms that flows do yield powerful invariants that get to the heart of protocol correctness. The running time for the final abstract model was around 5.5 hours.

D. Flows and State Explosion

A surprising discovery during this proof process was that, even after we have chosen important flows involving the directory, using all the flows is not the right thing to do as it leads to state explosion in the abstract model. Apart from various flows for requesting shared and exclusive accesses we had a collection of flows for write backs and invalidates. Unlike the former flows the latter flows were not incompatible with themselves, that is, there could well be many of these flows alive at the same time. Thus, in the abstract model, the $Other$ process can fire rules from these flows multiple times and saturate the auxiliary variables leading to an explosion in the number of states. It took us some time to understand this phenomenon, especially since augmenting the concrete model

with auxiliary variables increased the state space by at most a factor of 2. That is, the auxiliary variables don't increase the number of states in the concrete model by much, they only widen the state. But this is not so for the abstract model, especially if we have flows that don't have too many conflict constraints. This experience indicates that flows that appear in their own conflict sets might be best ones to use.

E. Flows as Validation Collateral

Apart from helping us prove the safety properties of interest by yielding invariants, the flows themselves are valuable validation collateral. The CMP method not only uses lemmas but also validates them in the process. Thus, we are not only using flows but also proving them correct. To see the crucial flows of the protocol exercised and to have an assurance that important actions of the protocol, like the directory sending grant exclusive/shared access messages, happen precisely as specified by flows, constitutes a much stronger validation of the protocol than just verifying a final global property. In fact, the architects who saw our proofs were impressed more with the fact that we validated the flows than they were with the global safety properties that we verified!

VIII. Conclusion

Finding invariants is one of the central problems of formal verification and extensive research is being carried out to find invariants automatically or via user provided annotations. While the flow based technique falls into the latter category, it has the advantage that flows arise naturally and are readily available. Ideally, annotations (or user supplied information) should be 1) easy to find, 2) provide relevant information precisely and in an easy to understand manner and 3) stay stable as the design details change. Flows have all three properties which makes them so attractive to use.

Apart from message passing distributed systems handled here, flows or partial orders on system events can also be applied to other types of distributed system. Lamport [27] has used partial orders analogous to flows in reasoning about mutual exclusion algorithms. In [27] the partial order is defined over *operations* which consist of sequences of atomic events. In addition to the precedence relation over events used in this paper, Lamport also defines a *can influence* relation over operations. While we used flows to derive invariants, Lamport (manually) reasons directly in terms of the partial order to prove the mutual exclusion property (which is also defined in terms of the precedence relation). A natural extension to our work is to generalize the notion of flows along the lines suggested by [27] to address other types of distributed systems.

In this paper we used abstraction to verify properties of a given model. An important related problem is bridging the abstraction gap between two existing models, such as the logical specification of a coherence protocol and its RTL implementation. Chen and others [28], [29] have developed techniques that address this problem and another natural extension to our work is to investigate the applicability of these techniques to our protocols.

Acknowledgments We thank Ching-Tsun Chou for building and validating the original LCP Murphi model that we validated parametrically in this paper.

References

[1] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proc. FMCAD*, 2008.
[2] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Proc. CHARME*, 2001.
[3] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. FMCAD*, 2004.
[4] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," in *Automated Verification of Infinite State Systems*, 2005.
[5] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$ counter abstraction," in *Proc. CAV*, 2002.
[6] S. K. Lahiri and R. Bryant, "Constructing quantified invariants," in *Proc. TACAS*, 2004.
[7] A. E. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *Proc. CADE*, 2000.
[8] J. Bingham, "Automatic non-interference lemmas for parameterized model checking," in *Proc. FMCAD*, 2008.
[9] Y. Lv, H. Liu, and H. Pan, "Computing invariants for parameter abstraction," in *Proc. MEMOCODE*, 2007.
[10] S. Das, D. L. Dill, and S. Park, "Experience with predicate abstraction," in *Proc. CAV*, 1999.
[11] S. Park and D. L. Dill, "Verification of flash cache coherence protocol by aggregation of distributed transactions," in *Proc. SPAA*, 1996.
[12] M. Talupur, S. Krstic, J. O'Leary, and M. R. Tuttle, "Parameteric verification of industrial strength cache coherence protocols," in *Proc. Workshop on Designing Correct Circuits*, 2008.
[13] L. Seiler *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27(3), Aug. 2008.
[14] S. K. Lahiri and R. Bryant, "Indexed predicate discovery for unbounded system verification," in *Proc. CAV*, 2004.
[15] G. Delzanno, "Automated verification of cache coherence protocols," in *Proc. CAV*, 2000.
[16] P. Abdullah, A. Buojjani, B. Jonsson, and M. Nilsson, "Handling global conditions in parameterized verification," in *Proc. CAV*, 1999.
[17] P. Abdullah and B. Jonsson, "On the existence of network invariants for verifying parameterized systems," in *Correct Systems Design-Recent Insights and Advances*, 1999.
[18] A. Cohen and K. Namjoshi, "Local proofs for global safety properties," in *Proc. CAV*, 2007.
[19] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parmeterized verification," in *Proc. VMCAI*, 2006.
[20] ——, "Proving Ptolemy right: Environment abstraction principle for parameterized verification," in *Proc. TACAS*, 2008.
[21] A. E. Emerson and V. Kahlon, "Model checking large-scale and parameterized resource allocation systems," in *Proc. TACAS*, 2002.
[22] E. A. Emerson and K. S. Namjoshi, "Reasoning about rings," in *Proc. POPL*, 1995.
[23] E. Clarke, M. Talupur, T. Touilli, and H. Veith, "Verification by network decomposition," in *Proc. CONCUR*, 2004.
[24] M. Talupur, "Abstraction Techniques for Infinite State Verification," Ph.D. dissertation, SCS, CMU, 2006.
[25] J. Rushby, "Verification diagrams revisited: Disjunctive invariants for easy verification," in *Proc. CAV*, 2000.
[26] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Proc. CHARME*, 1999.
[27] L. Lamport, "A new approach to proving correctness of multiprocess programs," *ACM TOPLAS*, vol. 1(1), pp. 84–97, 1979.
[28] X. Chen, S. German, and G. Gopalakrishnan, "Transaction based modeling and verification of hardware protocols," in *Proc. FMCAD*, 2007.
[29] X. Chen, Y. Yang, M. DeLisi, G. Gopalakrishnan, and C.-T. Chou, "Hierarchical cache coherence protocol verification one level at a time through assume guarantee," in *Proc. HLDVT*, 2007.

Industrial Strength Refinement Checking

Jesse Bingham*, John Erickson†, Gaurav Singh‡, and Flemming Andersen§
Intel Corporation,
*Email: jesse.d.bingham@intel.com, †Email: john.erickson@intel.com,
‡Email: gaurav.2.singh@intel.com, §Email: flemming.l.andersen@intel.com.

***Abstract*—This paper discusses a methodology used on an industrial hardware development project to validate various cache-coherence protocol components. The idea is to use a high level model (HLM) written in Murphi for model checking purposes, and then to use the HLM as a checker during dynamic (i.e. simulation based-) validation of the RTL. Such a checker requires a formal notion of what it means for the RTL to implement the HLM. Due to RTL pipelining, concurrency, and different RTL/HLM semantics, an appropriate notion is non-obvious. We employ a notion we call *behavioral refinement*, and describe a methodology for creating *refinement checkers*. A novel aspect of our methodology is that all "ingredients" are specified using System Verilog (SV): even the Murphi model itself is compiled into SV. Thus any off-the-shelf SV simulation engine can be used. We report the successful use of our refinement checkers to catch bugs in a real project at Intel and give an example illustrating our methodology.**

I. Introduction

A commonly used approach in verifying and validating complex hardware components involves constructing a *high level model* (HLM) of the component (e.g. [4], [8], [3], [13]). The HLM is a simplified version of the component that still models key functionality. For complex protocols such as cache coherency, the HLM is often written in a nondeterministic guarded-command language, such as Murphi [8]. Model checking small configurations of the HLM is then used to establish correctness of the protocol. Unfortunately, the HLM is too high-level to be synthesized into a high performance pipelined circuit, so a manually-written *register transfer level* (RTL) description is still the primary way of specifying the real hardware. This leaves open the question of whether or not the RTL is consistent with the HLM that was verified.

In this paper, we present a definition of what *implements* means in our context, which is based on the notion of *refinement* [2]. Our notion is tailored to the particular style of HLM (nondeterministic guarded commands) and implementation (clocked RTL), and is interesting in that it allows the HLM to take any finite number of steps per RTL clock; this was necessary both for the toy example given in this paper and the real industrial design we worked on. Many previous papers that allow for this are about super-scalar microprocessor verification, wherein the HLM must execute (up to) the instruction fetch width of the implementation. A few previous works involving HLMs that are similar to ours have identified a need for this allowance [7], [15].

We leave formal verification of this refinement relation as future work, since the techniques are not mature enough to be applied in a predictable way to a real industrial project that is constrained by strict schedules. However, we believe that *if you can't check it, you surely can't verify it*. By *check* we mean to watch for refinement violations during dynamic simulation, a technique we call *refinement checking*. In this light, we pursued refinement checking with the distinct goals of 1) evaluating the level of difficulty in writing the requisite refinement mapping, and 2) catching bugs during RTL development. We concluded that (1) is nontrivial, but certainly not prohibitively difficult. We were also successful at (2) during the few short months between bringing one of our refinement checkers online and writing this paper. We believe this is the first detailed report of using refinement checking on a industrial hardware design project *during development*[1].

One interesting aspect of our methodology is that all ingredients are ultimately written in System Verilog (SV). Hence any off-the-shelf SV simulator can be employed; there is no need to link the simulator with a model checker as was done in previous work [15]. To facilitate this, we developed a translator from Murphi code to SV called *mu2sv*. Mu2sv does a straightforward translation that maps each Murphi rule R to an SV function R`_sv` that take a record of type `MURPHI_STATE` as well as parameters for the rulesets.[2] R`_sv` returns another `MURPHI_STATE` that corresponds to the result of firing the rule.

II. Refinement Checking

Our goal is to monitor the RTL during simulation and flag an error if the behavior is not allowed by the Murphi HLM. In order to do so, we first must define what this means. Following that we present our methodology, and then conclude this section with a detailed example.

In Fig. 2 we pictorially define behavioral refinement, a notion that is relative to a given *refinement map* (RM). A RM is simply a many-to-one function that takes an RTL state and returns an HLM state. Given such a map, the RTL behaviorally refines the HLM if for any RTL behavior (shown across the bottom of the Fig. 2), there exists an HLM behavior that includes all RTL states mapped through the RM as shown. The figure shows that behavioral refinement allows each RTL clock to correspond to 0, 1, or more Murphi rules firing. Implicit in Fig. 2 is the fact that the first RTL state (the first state after reset in practice) must map to an *initial* `MURPHI_STATE`.

[1]The work of Tasiran et al. [15], though compelling, was done after the design was done and didn't catch any bugs.

[2]The `MUPRHI_STATE` SV type declaration is automatically generated by mu2sv and has a field for each variable in the Murphi model

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

```
type -------- Type Declarations --------------
CacheIndex : 0..1023;
CacheEntry : record
   State : enum {Invalid, Dirty, Clean};
   Addr  : ADDR;
   Data  : DATA;   end;

var ---- State Variable Declarations ---------
CacheArray : array [CacheIndex] of CacheEntry;
Cpu2Cache: Cpu2Cache_t;
Cache2Mem: Cache2Mem_t;

------ Rules (a.k.a Guarded Commands) --------
Ruleset i : CacheIndex "RecvStore"
  (Cpu2Cache.opcode = Store &
   CacheArray[i].State != Invalid &
   CacheArray[i].Addr = Cpu2Cache.Addr) |
  ( (forall j : CacheIndex :
      CacheArray[j].Addr != Cpu2Cache.Addr |
      CacheArray[j].State = Invalid) &
      CacheArray[i].State = Invalid ) ==>
  CacheArray[i].Data := Cpu2Cache.Data;
  CacheArray[i].State := Dirty;
  Absorb(Cpu2Cache);
end

Ruleset i : CacheIndex "Evict"
  CacheArray[i].State != Invalid ==>
  if (CacheArray[i].State == Dirty) begin
    Cache2Mem.opcode := WriteBack;
    Cache2Mem.Addr = CacheArray[i].Addr;
    Cache2Mem.Data = CacheArray[i].Data;
  end;
  CacheArray[i].State := Invalid;
end
```

Fig. 1. Murphi Code for Toy Cache Controller Example

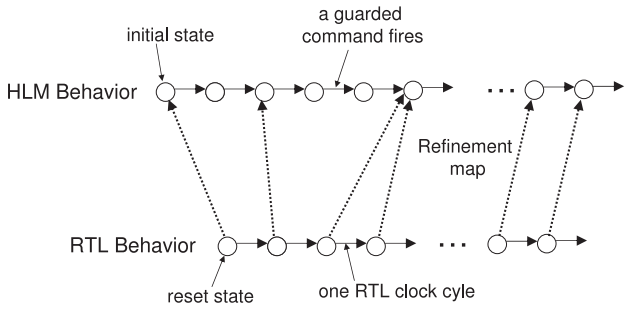


Fig. 2. Behavioral Refinement

We generalize this notion modestly by allowing the refinement function to depend on so-called *history variables* [2]. History variables are auxiliary variables added to the RTL that have no affect on RTL behavior, but rather record information about the past. In practice, history variables are extremely useful for writing a RM for pipelined RTL. We call an RTL state with history variables simply an *augmented RTL state*.

Refinement Checking Methodology. A refinement checker observes an RTL behavior (driven by some test stimuli), and attempts to construct a corresponding Murphi behavior that it behaviorally refines. If it is ever unable to do so, an error is flagged. This requires two distinct "ingredients" to be written by the human: the RM and *rule selection*. In our methodology, both of these are written in SV. Though mathematically `RM()` is a function that takes the current RTL state `r`, as an SV function it takes no parameters, but rather looks at RTL design

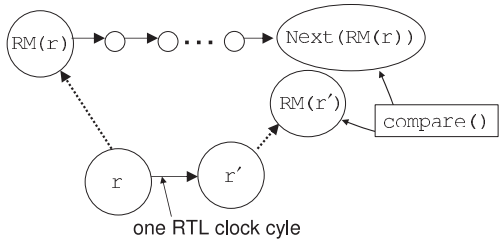


Fig. 3. Commutative diagram showing how our refinement checkers work.

signals internally via absolute signal names. Rule selection determines which sequence of rules to fire corresponding to each RTL clock cycle. When the Murphi rules are rulesets, rule selection must also determine the actual parameter values to use. In SV, rule selection is done by a function `Next(MURPHI_STATE ms)` that contains sequential code subjected to the restrictions that 1) all return statements in `Next()` return `ms`, and 2) all assignments in `Next()` having `ms` on the LHS have RHS of the form $R\_\mathtt{sv}(\mathtt{ms}, \ldots)$ (where $\ldots$ are the ruleset actual parameters). Since $R\_\mathtt{sv}$ errors out if invoked on `ms` for which R's guard is false, our restrictions imply that `Next(ms)` returns a `MURPHI_STATE` formed by firing *some* rule sequence from `ms`.

Fig. 3 shows how the checker operates using a commutative diagram. The current augmented RTL state `r` is first mapped through `RM` and then through the rule selection function `Next`. Then an error is raised if either during computation of `Next(RM(r))`, some $R\_\mathtt{sv}$ function is invoked when the guard is false, or if $\neg(\mathtt{Next}(\mathtt{RM}(\mathtt{r})) == \mathtt{RM}(\mathtt{r}'))$, where $\mathtt{r}'$ is the augmented RTL state in the next clock. A simply inductive argument shows that if the checker never throws an error, then there exists a Murphi behavior that the RTL simulation behaviorally refines.

Example: Toy Cache Controller. We now demonstrate these ideas through a toy example cache controller (CC). Murphi code for CC (abridged somewhat) is in Fig. 1. The state variables are:[3]

- `CacheArray` is an array of `CacheEntry`, which has fields State which can be `Invalid`, `Clean`, or `Dirty`, and a field each for the address and data.
- `Cpu2Cache` is a message being sent from the CPU to the Cache. To save space we omit the typedef.
- `Cache2Mem` is a message being sent from the Cache to main memory. To save space we omit the typedef.

Two rulesets are shown in Fig. 1. Both are parameterized by `i`, which is an index into `CacheArray`. `RecvStore` fires when a store command from the CPU is processed. The `Evict` ruleset is very non-deterministic in the sense that any valid line `i` can be evicted at any time. This is typical of non-deterministic HLMs; any RTL implementation will use some possibly complex eviction policy, but this complexity is abstracted away with non-determinism in the HLM. An RTL implementation is shown schematically in Fig. 4(a). This

[3]We note that a real design would have messages going from the cache to CPU and from memory to the cache.

implementation adds several details: deterministic eviction logic, two stages of pipelining, physically separate data and state/address arrays that get updated in different cycles.

Fig. 4(b) takes us through five consecutive RTL states with back-to-back store requests arriving on the `Cpu2Cache` interface, the first one requiring an eviction. Symbols A_i and D_i are used to denote addresses and data values, respectively. In cycle 1 the first store request (storing data D_0 to address A_0) arrives. Fig. 4(b) also shows the initial contents of the cache, which has A_1 dirty with data D_1, and A_2 clean with data D_2. In cycle 2 the first store is staged in the first pipestage, and the second store with address A_2 and data D_2' arrives. Cycle 3 has the first store causing the eviction of A_1 from the cache, and the writeback is staged in pipestage 2. Also A_0 and `Dirty` are written into the evicted line. The corresponding data D_0 is written into the data array in cycle 4 and A_2 is made dirty. Finally in cycle 5 the new data D_2' is written into the cache for A_2. Clearly three rules have fired in this example flow, i.e. two instances of `RecvStores` and one of `Evict`. However, unlike the HLM, they are *non-atomic* (take multiple clocks to complete), and exhibit *true concurrency* (several rules can execute at the same time).

Because different data structures are updated at different cycles (i.e. non-atomicity), RMs must generally sample RTL signals at different temporal offsets from the current cycle. Ultimately this is done using history variables only. For CC, the RM must account for the fact that a store, which happens atomically in Murphi, takes three cycles in the RTL. Hence the map samples the incoming `Cpu2Cache` message, the state/address array, and the data array in consecutive clock cycles. Similarly the outgoing `Cache2Cpu` message is sampled in the same cycle that the data array is. The resulting RM is shown in Fig. 5, in which $signal@k$ is the value of $signal$, temporally offset by k cycles. In practice, we shift the RM so that all offsets are non-positive and thus implementable with history variables.

Rule selection deals with resolving true concurrency in the RTL. As discussed above, rule selection is encapsulated in a user-written SV function `Next` , which takes the current Murphi model state `ms`, and applies some sequence of rules to `ms`. This function for CC is shown in Fig. 5. Here we see that if there was a valid message 2 cycles earlier on the `Cpu2Cache` interface, then we calculate the target cache entry `i`. If an eviction is needed then we apply the appropriate `Evict` rule.[4] Finally we do `RecvStore`. In a realistic example there would be additional cases, e.g. for LOAD commands. Note that Fig. 5 results in either 0, 1, or 2 rules being fired per RTL clock. Fig. 4(c) shows the Murphi behavior corresponding to the RTL behavior of Fig. 4(b) and how they are connected by the RM.

III. Related Work

The most closely related work is that of Tasiran et al. [15]. Here the HLM was written in TLA+ and during simulation pairs of consecutive HLM states are passed off to the TLC model checker which checks if they are a valid HLM transition. Though not explicitly discussed in the paper, Tasiran et al. also allow multiple HLM steps per RTL clock; we feel that this is an important feature that deserves highlighting. Since their RM was expressed using 8K lines of C++ code, it is unlikely that one could use formal tools to reason about it, i.e. to prove that refinement holds. Contrarily, since our RM is expressed as synthesizable SV code, our approach is much more amenable to formal reasoning using standard tools.

```
function MURPHI_STATE Refinement_Map();
  MURPHI_STATE ms;
  for (int i = 0; i < CACHE_LINES; i++) begin
     ms.CacheArray[i].State = RTL.AddrArray[i].State@0;
     ms.CacheArray[i].Addr  = RTL.AddrArray[i].Addr@0;
     ms.CacheArray[i].Data  = RTL.DataArray[i]@+1;     end;
  ms.Cpu2Cache = RTL.Cpu2Cache@-1;
  ms.Cache2Cpu = RTL.Cache2Cpu@+1;
  return(ms); end;

function MURPHI_STATE Next(MURPHI_STATE ms);
  if (RTL.Cpu2Cache.Valid@-2) begin
     i = get_target_cache_index();
     if (eviction_needed()) ms = Evict_sv(ms,i);
     if (RTL.Cpu2Cache.Opcode@-2 = STORE)
        ms = RecvStore_sv(ms,i); end;
  return(ms); end;
```

Fig. 5. Cache Controller Refinement Map and Next function (rule selection)

Chen et al. use a form of Murphi called *Hardware Murphi* to verify hardware protocols [6]. It involves combining a traditional Murphi model with another model that specifies signal and timing info that can be used to generate VHDL. Other work includes compiling LTL or other formal assertions into dynamic checkers (e.g. [14]). These assertions involve RTL signals so no RM is necessary.

There have been a plethora of definitions for what it means for a lower-level system S_1 to implement a higher level system S_2. The importance of allowing S_2 to stutter was first promoted by Lamport [10]. Later Lamport and Abadi [2] showed that under certain assumptions, RMs can be shown to always exist, however their formalism does not allow the HLM to take multiple steps for each implementation step. To our knowledge, there are only a few definitions that allow for this, and many are in the context of superscalar microprocessor verification. These typically verify the existence of $t \in \{0, \ldots, k\}$ such that t HLM steps matches an implementation step. The bound k is the instruction fetch width of the implementation. For more on refinement for microprocessor verification see e.g. [1], [12].[5]

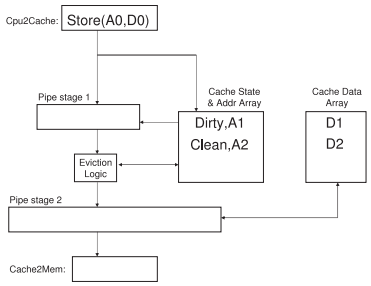
Finally, our work advocates manually writing a RM, rather than using more automated techniques such as flushing [5]. Like other work on RMs for real industrial RTL designs [9], [15], we have found manually-written maps more appropriate.

IV. Case Study

Our case study was a hierarchical cache protocol; the offchip protocol was Intel's QPI [11], while on-chip coherence is managed by a proprietary protocol that we'll simply call *Level-1 protocol (L1P)*. Our work was divided into two main parts:

[4]The functions `get_target_cache_index()` and `eviction_needed()`, not shown, are user-written based on knowledge of RTL details.

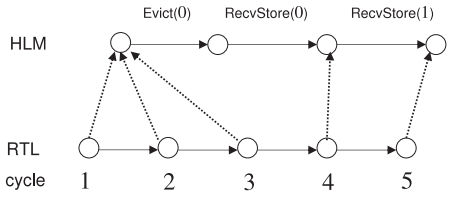
[5]We note, however, that verification of protocol implementations and that of microprocessor implementations are different beasts since the former operate in very restricted environments and have much more concurrency .



(a) RTL

cycle	1	2	3	4	5
Cpu2Cache	$Store(A_0, D_0)$	$Store(A_2, D_2')$			
Pipestage1		$Store(A_0, D_0)$	$Store(A_2, D_2')$		
Pipestage2			$Store(A_0, D_0)$ $WB(A_1, D_1)$	$Store(A_2, D_2')$	
State/Addr Array	$Dirty, A_1$ $Clean, A_2$	$Dirty, A_1$ $Clean, A_2$	$Dirty, A_0$ $Dirty, A_2$	$Dirty, A_0$ $Dirty, A_2$	$Dirty, A_0$ $Dirty, A_2$
Data Array	D_1 D_2	D_1 D_2	D_1 D_2	D_0 D_2	D_1 D_2'
Cache2Mem				$WB(A_1, D_1)$	

(b) Five consecutive RTL states



(c) Behavioral Refinement for behavior of Fig. 4(b).

Fig. 4. Cache Controller Example

first, building the QPI/L1P Murphi model and model checking it, then building refinement checkers for two key protocol components and using them during dynamic RTL simulation. These two checkers are now discussed

L1PM Refinement Checker. The central hub of L1P that handles processor core memory requests and also interfaces to QPI is a component called the *L1P master* (L1PM). We used our Murphi HLM and the methodology of this paper to create a refinement checker for L1PM. Developing the L1PM refinement checker took about 3 months of person effort, after the HLM and *mu2sv* had already been developed. The manual effort involved is developing and debugging the RM and rule selection. It was integrated into the validation test environment and run alongside the standard checkers and assertions for dynamic validation. After only one month, 8 RTL bugs had been found using the L1PM refinement checker.

QPI Home Agent Refinement Checker. The QPI Home Agent (HA) is the part of the memory controller that receives requests, sends snoops, and generally manages coherency [11]. Though at the time of writing this paper the QPI HA refinement checker was not yet deployed to the validation team, it is worth mentioning since it is considerably more complex than the L1PM checker. The checker involves seven different Murphi rulesets . Rule selection can yield up to 8 rule instances firing per clock cycle. The RM involves signals from five different pipestages and uses a window three clocks wide for temporal offset sampling. The HA refinement checker is still under development, but it passes a test suite of around 200 tests designed to exercise basic functionality.

V. Conclusions and Future Work

Though we believe refinement checking is an exciting approach to dynamic validation, ultimately we would like to *formally prove* that the refinement holds. Clearly, formal proof of behavioral refinement can be achieved by proving that the refinement checker never raises an error. However, we note that since all "ingredients" needed for refinement checking are required also for formal proof (except the test stimuli), and furthermore refinement checker development is a far less capricious activity, we advocate *starting* with checking and pushing towards formal proof only once the checker is in place.

References

[1] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for superscalar microprocessor correctness statements. *Software Tools for Technology Transfer*, 4(3):298–312, 2003.

[2] M. Abadi and L. Lamport. On the existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[3] M. Azimi, C.-T. Chou, A. Kumar, V. W. Lee, P. K. Mannava, and S. Park. Experience with applying formal methods to protocol specification and system architecture. *J. Formal Methods in System Design*, 22(2), 2003.

[4] R. Beers. Pre-RTL formal verification: an intel experience. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 806–811, 2008.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *6th Intl. Conf. on Computer-Aided Verification (CAV)*, pages 68–80, 1994.

[6] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[7] N. Dave, M. C. Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'05)*, July 2005.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[9] R. Kaivola. Formal verification of pentium ® 4 components with symbolic simulation and inductive invariants. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 170–184, 2005.

[10] L. Lamport. What good is temporal logic? *Information Processing*, pages 657–668, 1983.

[11] R. A. Maddox, G. Singh, and R. J. Safranek. The intel quickpath interconnect architecture. *Dr. Dobb's Journal*, May 19 2009.

[12] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for pipelined machines using WEB refinement. *ACM Trans. on Design Automation of Electronic Systems*.

[13] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, 2000.

[14] K. Ng, A. J. Hu, and J. Yang. Generating monitor circuits for simulation-friendly gste assertion graphs. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 409–416, 2004.

[15] S. Tasiran, Y. Yu, and B. Batson. Linking simulation with formal verification at a higher level. *Design and Test of Computers, IEEE*, 21(6):472–482, 2004.

Towards A Formally Verified Network-on-Chip

Tom van den Broek and Julien Schmaltz
Institute for Computing and Information Sciences
Radboud University Nijmegen, The Netherlands
Email: tombroek@science.ru.nl,julien@cs.ru.nl

***Abstract*—Multi-Processor Systems-on-Chip (MPSoC) designs are constructed by assembling pre-designed parameterized components. Communications are crucial to their overall functionality and performance. Formal verification methods have been intensively applied to processing elements, e.g., microprocessors. Very little work has been done with respect to communication modules. We present the formal specification of a packet switched NoC and its proven refinement. At the specification level, routing decisions are computed *at once* before packets get injected in the network. In the implementation, routing decisions are distributed over each individual node. We prove that the implementation behaves according to its specification for a 2D-mesh NoC. All models and proofs have been checked using the ACL2 theorem proving system. To the best of our knowledge, this work constitutes the first cross-layer verification of on-chip communication networks.**

I. Introduction

Formal verification often consists in showing that for every execution of an implementation there exists an execution of its abstract specification with the same visible effects. This approach has been successfully applied to processing elements (e.g. microprocessors [7], [8]). Multi-Processor Systems-on-Chip designs offer increased performance by combining several processing and memory cores on a single die. The interconnect is becoming crucial to the overall functionality of an MPSoC [13]. When the number of interconnected units grows, bus performances decrease. Networks-on-Chips (NoCs) [2] is a solution that could meet future system performance.

Regarding buses, the recent work of Böhm and Melham is the only effort trying to fill the gap between abstract specifications and low level implementations [3]. Previous efforts concentrate on proving properties on low-level implementations using model-checking [11] or combination of model-checking with theorem proving [1]. Gebremichael *et al.* [5] provide a parametric analysis of part of the AEthereal NoC [6]. All these works considers *implementations* only. Regarding specifications, Schmaltz *et al.* [12], [4] propose a generic network model, named GeNoC. We present models that are variations of the GeNoC model. Each model is at a different abstraction level. Our contribution is a formal relation between instances of these models.

Our goal is to provide a methodology to support the abstract specification of NoCs and the proof that implementations conform to it. In this paper, we present an initial effort towards this goal. We present the formal specification of a packet switched NoC. At the specification level, routing decisions are computed *at once* before packets get injected in the network. In the implementation, routing decisions are distributed over each individual node, i.e., hop-by-hop. Details of the implementation model are available [14]. The original contribution of this paper consists of the definition of the specification model and the proof that the implementation behaves according to its specification for a 2D-mesh network based on the HERMES NoC [10]. All models and proofs have been checked using the ACL2 theorem proving system [9] and are available on the web[1]. To the best of our knowledge, this paper presents the first cross-layer verification of a NoC.

II. A NoC Example: HERMES

HERMES [10] is based on a 2D mesh architecture (Fig. 1). Each node is made of an IP core and a switch. Each switch has five bi-directional ports: East, West, North, South connecting to the neighbor switches, and Local to the IP core.

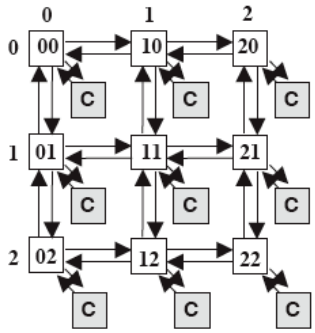


Fig. 1. Mesh architecture [10]

The routing policy is based on a deterministic, minimal algorithm: the *XY routing algorithm*. Each packet is routed on one dimension at a time. It travels first along the X axis until the first coordinate of the destination is reached, and then travels along the Y axis. This algorithm is recalled in Fig. 2.

Our HERMES instance uses *packet switching* (store-and-forward). A packet constitutes the fixed size basic unit, which travels in the network. A packet contains a *header* with routing information and a payload with data. A packet is sent autonomously through the network.

Each port of the switch has an input buffer queue. A Round Robin priority policy is used to access the output ports. If the requested port is busy, packets are blocked and the request signal remains active. The transmission between two different nodes is ruled by an handshake protocol.

[1] www.cs.ru.nl/∼julien/Julien_at_Nijmegen/FMCAD09.html

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

```
XYRouting(from,to) :
if from=to     /* destination reached */
then take the Local port
else
  if Xfrom != Xto
  then     /* change X */
       if Xfrom < Xto
       then take port East
       else take port West
  else     /* change Y */
       if Yfrom < Yto
       then take port South
       else take port North
```

Fig. 2. XY routing algorithm

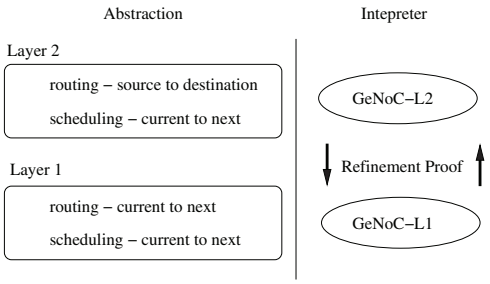


Fig. 3. Verification Approach

III. Verification Approach

A. NoC Abstraction, Interpreter, and Property

Our method consists of a NoC model, two abstraction levels for that model, and a NoC interpreter for each level. Both interpreters are simulators for networks composed of identical nodes following a generic router model. They are proven equivalent (Section VII). The generic model has a few design-dependent functions which constitute the user-input. These functions are the Link Layer protocol (e.g., handshake), the routing logic (e.g., XY), and the scheduling policy (e.g., packet switching). Together with the router model, they are described below. The main difference between the interpreters lies in the routing decisions. The specification (Layer 2) supports *source* routing, where routes are computed before sending packets. The implementation supports *distributed* routing where each node computes the next step in a route.

B. Network Model

We assume a generic architecture composed of an arbitrary – but finite – number of nodes and a finite number of connections between any two nodes. Each node is uniquely identified by its position. A node includes a local memory and a router. A router is defined by a set of ports and four functions: input and output units, routing control, and flow control (see Fig. 4). All nodes are identical.

The main elements of a port are the data and control signals, and internal buffers (Fig. 4). Formally, a port is a tuple $\langle addr, stat, data, buff \rangle$, where $addr$ is a unique address, $stat$ stores the values of the control signals and other state

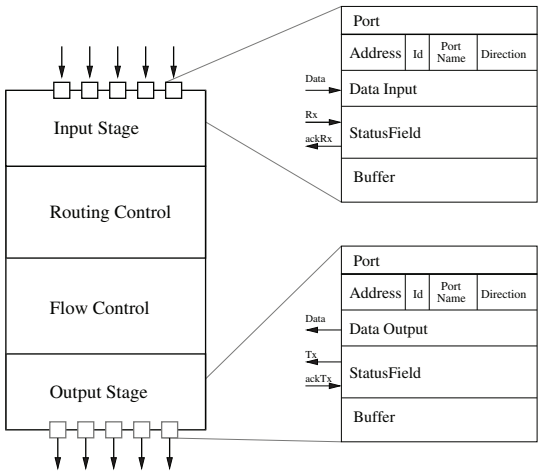


Fig. 4. A router and its ports

components, $data$ denotes the values of the data signals, and $buff$ represents the value of the buffers.

An address is a tuple $\langle coor, pid, dir \rangle$, where $coor$ is the unique identifier of the node the port belongs to, pid the name of the port (e.g., *west*, *south*), and dir the direction, i.e., 'i' for an input port or 'o' for an output port. The topology is a list where each element is a pair of port addresses (p_i, p_j), which means that port p_i is connected to port p_j. A node is defined as the set of ports, where the address of each port p is the same. These ports define the state of the node. The set of all ports of a network defines the state of the network.

Functions `ProcessInputs` and `ProcessOutputs` define the low level protocols which use the control signals to transfer the content of the data signals to the internal buffers in case of an input port, or to transfer the content of the buffers to the data signals in case of an output port.

Function `RouteControl` applies the routing logic to one or more ports of a node. It returns a list of *routed ports*, i.e., ports together with routing information. The only design-dependent function is function `routing-logic` which implements the routing algorithm.

Function `FlowControl` implements the switching technique, e.g., packet, circuit, or wormhole. In case of conflict, this function also resolves priorities. This function extracts from the routed ports the packets that are ready to be transmitted. The only design-dependent function is function `switch-ports` which effectively schedules packets. Those scheduled packets are moved to the output ports computed by the routing control function.

All these functions form function *router* (Fig. 5), which updates a node state. Note that a node is equipped with a memory which is available to each port and each function. Argument `nstmem` represents that memory. To simplify the presentation, we assume that such a memory element is given as input argument of any function that accesses it. This argument is not explicitly mentioned any further.

```
router(nst,nstmem) :
let (nst nstmem) be
    RouteControl(ProcessInputs(nst), nstmem)
    in
    let (nst nstmem) be
        FlowControl(nst, nstmem)
        in
        return ProcessOutputs(nst), nstmem
```

Fig. 5. Function router

IV. Network Model Instance – User Input

For the HERMES 2D-mesh, an address is defined by the xy-coordinate of the node, $pid \in \{n, s, e, w\}$, and $dir \in \{i, o\}$. A topology element identifies bi-directional links between two nodes, for instance $(\langle 0\ 0, e\rangle, \langle 1\ 0, w\rangle)$.

The input and output units implement an handshake protocol. A node can request the transmission using signal `Tx` connected to signal `Rx` of the receiver node. The latter can deny or grant the access using signal `AckRx` connected to signal `AckTx` of the sender.

Function `routing-logic` is defined to implement the routing algorithm described in Fig. 2.

Fig. 6 shows the instantiation of function `switch-ports` for packet switching, named `pkt-switch-ports`. It takes as arguments the list of the output ports (`outports`); an input port (`from`), the content of which has been routed; and the state of the node (`nst`). Function `pkt-switch-ports` finds the output to which the input port must be connected, and checks whether this port can accept the packet. Each node has a one-place output buffer for each output port. A port accepts a packet if its buffer is empty. If such a port exists, function `switchBuffer` transfers the content of the input port to the output port, i.e., loads the output port and clears the input port.

```
pkt-switch-port(outports, from, nst)
let to be outports[0]
    in
    case
       outports = null: return nst

       status-route(port-status(from))
       and not (port-bufferFull(to)):
       return switchBuffer(nst,from,to)

       default:
       pkt-switch-port(outports[1..]), from,nst)
```

Fig. 6. Function pkt-switch-port

V. Specification – GeNoC-L2

Function *GeNoC*-L2 (Fig. 7) is the core function of our interpreter for the specification layer. Input argument `simL` defines the length of the simulation. It takes as additional arguments the set of packets to be sent (`m`), the current state of the network (`ntkst`), an accumulator of packets that have reached their destination (`arr`, initially empty), the current simulation step (`z`, initially 0), and the topology (`topo`). It returns the list of arrived packets, the list of delayed packets, and the state of the network at the end of the simulation.

```
GeNoC-L2(m, ntkst, arr, z, topo, simL) :
if simL = 0 return arr,m,ntkst
else
 let (dep, del) be
     depart-L2(ntkst, m, z)
     in
     let newntkst be
         step-ntk-L2(dep, topo)
         in
         GeNoC-L2(del,newntkst, add(z,arr),
                  z + 1,topo, simL-1)
```

Fig. 7. Function GeNoC-L2

Function `depart-L2` controls packet injection. According to a user-defined criterion, it determines which packets can be in the network. At the specification level, this function uses *source routing* and appends to a packet its route *from its source to its destination*. It inserts this extended packet in the local input port of its source node. It returns a new state (`dep`) and a list of delayed packets (`del`). Function `step-ntk-L2` (see below) actually performs one simulation step. It updates the global network state. Those packets that are at their destination are extracted from this new state and appended to accumulator *arrived*. The next recursive call processes the delayed packets, the new network state, and time is incremented by 1.

```
step-ntk1-L2(ntslist, ntkst):
if ntslist = null return ntkst
else
let newnst be ProcessInputs-L2(ntslist) in
    let (newnstlist,newntkmem1) be
        RouteControl-L2(newnstlist,ntkmem) in
        let (newnstlist, newntkmem) be
            FlowControl-L2(newnstlist, newntkmem1)
            in
            return ProcessOutputs-L2(newnstlist),
                   newntkmem

step-ntk-L2(ntkst, topo):
let newntkst be
    step-ntk1-L2 (ports-nodelist(ntkst), ntkst)
    in
    updateNeighbours-L2(newntkst,topo)
```

Fig. 8. Function step-ntk-L2

At the specification level, every element of our router is applied sequentially over all nodes. Function `step-ntk1-L2` (Fig. 8) takes as arguments a list of nodes to be processed (`ntslist`) and the current network state (`ntkst`). It updates the network state. Function `ProcessInputs-L2` applies the input stage to all nodes. As routes have been computed at injection time, the RouteControl module simply "reads" the next hop as the first element of the route. Function `RouteControl-L2` performs this "routing" decision at all nodes. After that, function `FlowControl-L2` applies the packet switching policy to all nodes. Finally, function `ProcessOutputs-L2` applies the output stage at all nodes.

Function `step-ntk` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntk1-L2`. Function `updateNeighbours-L2` simulates the transfer of data from output data signals to input data signals. This function removes the first element of routes.

VI. Implementation – GeNoC-L1

Function $GeNoC$-L1 is the core function of our interpreter for the implementation layer. It takes and returns the same arguments as $GeNoC$-L2. Its definition is obtained by replacing every occurrence of "L2" with "L1" in Fig. 7. It only injects a packet without appending any additional information.

```
step-ntk1-L1(ntslist, ntkst):
if ntslist = null return ntkst else
let newnst be router(ntslist[0]) in
    let newntkst be
        step-ntk1-L1(ntslist[1..], ntkst) in
        return ports-update(newntkst,newnst)
step-ntk-L1(ntkst, topo):
let newntkst be
    step-ntk1-L1 (ports-nodelist(ntkst), ntkst) in
    updateNeighbours-L1(newntkst,topo)
```

Fig. 9. Function `step-ntk-L1`

Function `step-ntk-L1` (Fig. 9) is based on recursive function `step-ntk1-L1`. The latter takes as arguments a list of nodes to be processed (`ntslist`) and the current network state (`ntkst`). It updates the network state. For each node, it applies function `router`. Function `ports-update` effectively updates the state of the nodes. Finally, function `step-ntk-L1` extracts the node structures from the list of ports (function `ports-nodelist`), and calls `step-ntk1-L1`. Function `updateNeighbours-L1` simulates the transfer of data from output to input signals.

VII. Equivalence Proof

The theorem connecting the two models is shown in Fig. 10. Function `transform` simply removes all routes from extended packets. This theorem states that after the application of `transform` the lists of arrived packets, the lists of packets still en route in the network, and the final network state produced by $GeNoC$-L2 equals those produced by $GeNoC$-L1. The proof in itself is nothing deep. The two interpreters manipulate the same functions. The only difference is in the ordering of these function calls. The difficulties lie in getting the right model definitions and the details of the formal proofs.

Our proof depends on three axioms about the topology and the state generated from it. They basically state that the initial network state is well-formed, e.g., it agrees with the topology.

VIII. Conclusion and Future Work

We presented the first effort in building a verification methodology of NoCs. We defined two abstractions layers and proved their equivalence. The source routing specification is correctly refined into a distributed routing implementation. A large part of our model and the proof is design-independent. Our plan is to extract the generic part of our proof and obtain a general verification method. We also are working on extending GeNoC-L2 to support global and application-independent properties like functional correctness or deadlock avoidance [15]. We are convinced that the structure of our implementation is similar to the actual structure of RTL designs. One has now to relate our algorithm to the RTL. We plan to investigate the generation RTL code from our models.

```
Theorem:
let (arr-l1, m-L1, ntkst-L1) be
    GeNoC-L1(m, ntkst, arr, z, topo, simL) in
    let (arr-l2, m-L2, ntkst-L2) be
        GeNoC-L2(m, ntkst, arr, z, topo, simL) in
        transform(arr-L2) = arr-L1 and
        m-L2 = m-L1 and
        transform(ntkst-L1) = ntkst-L2
```

Fig. 10. Equivalence theorem

Acknowledgments

This research is supported by NWO/EW project Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811.

References

[1] H. Amjad. Model Checking the AMBA Protocol in HOL. Technical report, University of Cambridge, Computer Laboratory, September 2004.

[2] L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.

[3] P. Böhm and T. Melham. A refinement approach to design and verification of on-chip communication protocols. In *Formal Methods in Computer-Aided Design (FMCAD'08)*. IEEE Computer Society, 2008.

[4] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A formal approach to the verification of networks on chip. *EURASIP Journal on Embedded Systems*, 2009(Article ID 548324):14 pages, 2009.

[5] B. Gebremichael, F. Vaandrager, and M. Zhang. Formal Models of Guaranteed and Best-Effort Services. Tech. Rep. Institute for Computing and Information Sciences, Radboud University Nijmegen, March 2005.

[6] K. Goossens, J. Dielissen, and A. Rădulescu. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5):414–421, September-October 2005.

[7] W. A. Hunt. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[8] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. *Computer-Aided Verification CAV*, LNCS 818, pages 68–80, Standford, California, USA, 1994. Springer.

[9] M. Kaufmann, P. Manolios, and J Strother Moore. *ACL2 Computer Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[10] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. HERMES: an infrastructure for low area overhead packet-switching networks on chip. *Integration*, 38(1):69–93, 2004.

[11] A. Roychoudhury, T. Mitra, and S.R. Karri. Using Formal Techniques to Debug the AMBA System-on-Chip Bus Protocol. In *Design Automation and Test Europe (DATE'03)*, pages 828–833, 2003.

[12] J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 20(3):239–348, 2008.

[13] G. Spirakis. Beyond Verification: Formal Methods in Design. In A. Hu and A.K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD'04)*, LNCS 3312, Austin, Texas, USA, November 2004. Springer-Verlag. Invited Speaker.

[14] T. van den Broek and J. Schmaltz. A generic implementation model for the verification of networks-on-chips. In *8th Intl. Workshop on the ACL2 Theorem Prover and Its Application*, pages 130–134, Northeastern Univ., Boston MA, USA, 2009. ACM.

[15] F. Verbeek and J. Schmaltz. Formal validation of deadlock prevention in networks-on-chips. In *8th Intl. Workshop on the ACL2 Theorem Prover and Its Application*, pages 135–145, Northeastern Univ., Boston MA, USA, May 11–12 2009. ACM.

Hardware/Software Co-verification of Cryptographic Algorithms using Cryptol

Levent Erkök Magnus Carlsson Adam Wick

Galois, Inc.
421 SW 6th Ave. Suite 300
Portland, OR 97204

***Abstract*—Cryptol is a programming language designed for specifying cryptographic algorithms. Despite its high-level modeling nature, Cryptol programs are fully executable. Further, a large subset of Cryptol can be automatically synthesized to hardware. To meet the inherent high-assurance requirements of cryptographic systems, Cryptol comes with a suite of formal-methods based tools that enable users to perform various program verification tasks. In this paper, we provide an overview of Cryptol and its verification toolset, especially focusing on the co-verification of third-party VHDL implementations against high-level Cryptol specifications. As a case study, we demonstrate the technique on two hand-written VHDL implementations of the Skein hash algorithm.**

***Index Terms*—Specification and Verification, Equivalence checking, HW/SW Co-verification, Cryptography**

I. Introduction

Cryptol is a domain specific language tailored for the domain of cryptographic algorithms.[1] Cryptol specifications are fully executable on commodity hardware using an interpreter, and a large subset of Cryptol programs can be automatically synthesized to various hardware platforms via translation through VHDL. Explicit support for verification is an indispensable part of the Cryptol toolset, due to the inherent high-assurance requirements of the application domain. To this end, Cryptol comes with a suite of formal-methods based tools, allowing users to perform various program verification tasks. In this paper, we provide a short overview of the Cryptol language, especially focusing on verification.

Cryptol is a pure functional language, built on top of a Hindley-Milner style polymorphic type system, extended with size-polymorphism and arithmetic type predicates [1]. The size-polymorphic type system has been designed to capture constraints that naturally arise in cryptographic specifications. To illustrate, consider the following text from the AES definition [2, Section 3.1]:

> The **input** and **output** for the AES algorithm each consist of **sequences of 128 bits**. ... The **Cipher Key** for the AES algorithm is a **sequence of 128, 192 or 256 bits**. Other input, output and Cipher Key lengths are not permitted by this standard.

This description is captured precisely in Cryptol by the following type:

```
{k} (k >= 2, 4 >= k)
      => ([128], [64*k]) -> [128]
```

Anything to the left of => are quantified type-variables and predicates on them. In this case, the type is size-polymorphic, relying on the size-variable `k`. The predicates constrain what values the quantified size-variables can accept: Here, `k` is restricted to be between 2 and 4. To the right of =>, we see the actual type. The input to the function is a pair of two words, the first of which is the 128-bit plain-text. The second argument is a `64*k`-bit wide word (i.e., 128, 192, or 256 bits, depending on `k`) representing the key. The output of the function, the ciphertext, is another 128-bit word. Note how this type precisely corresponds to the English description.

II. Verification of Cryptol Programs

Cryptol's program verification framework has been designed to address equivalence and safety checking problems.

The equivalence-checking problem asks whether two functions f and g agree on all inputs. Typically, f is a reference implementation of some algorithm, following a standard textbook style description, and g is a version optimized for time and/or space for a particular target platform. The equivalence-checking framework allows one to formally prove that f and g are semantically equivalent, ensuring that the (often very complicated and extensive) optimizations performed during synthesis are semantics preserving. Note that the final implementation (i.e., g) need not necessarily be in Cryptol: an important use case of the verification framework is in verifying third-party algorithm implementations (typically in VHDL) are functionally equivalent to their high-level Cryptol versions. In this case, Cryptol acts as a HW/SW co-verification tool.

The safety-checking problem is about run-time exceptions. Given a function f, we would like to know if f's execution can perform operations such as division-by-zero or index-out-of-bounds. These checks are essential for increasing reliability and availability of Cryptol-based products, since they eliminate the need for sophisticated run-time exception handling mechanisms whenever applicable.

The Cryptol toolset comes with a push-button equivalence/safety checking framework to answer these questions automatically for a large subset of the Cryptol language [3]. As the reader might suspect, the push-button system suffers

[1] Cryptol toolset licenses are freely available at www.cryptol.net.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

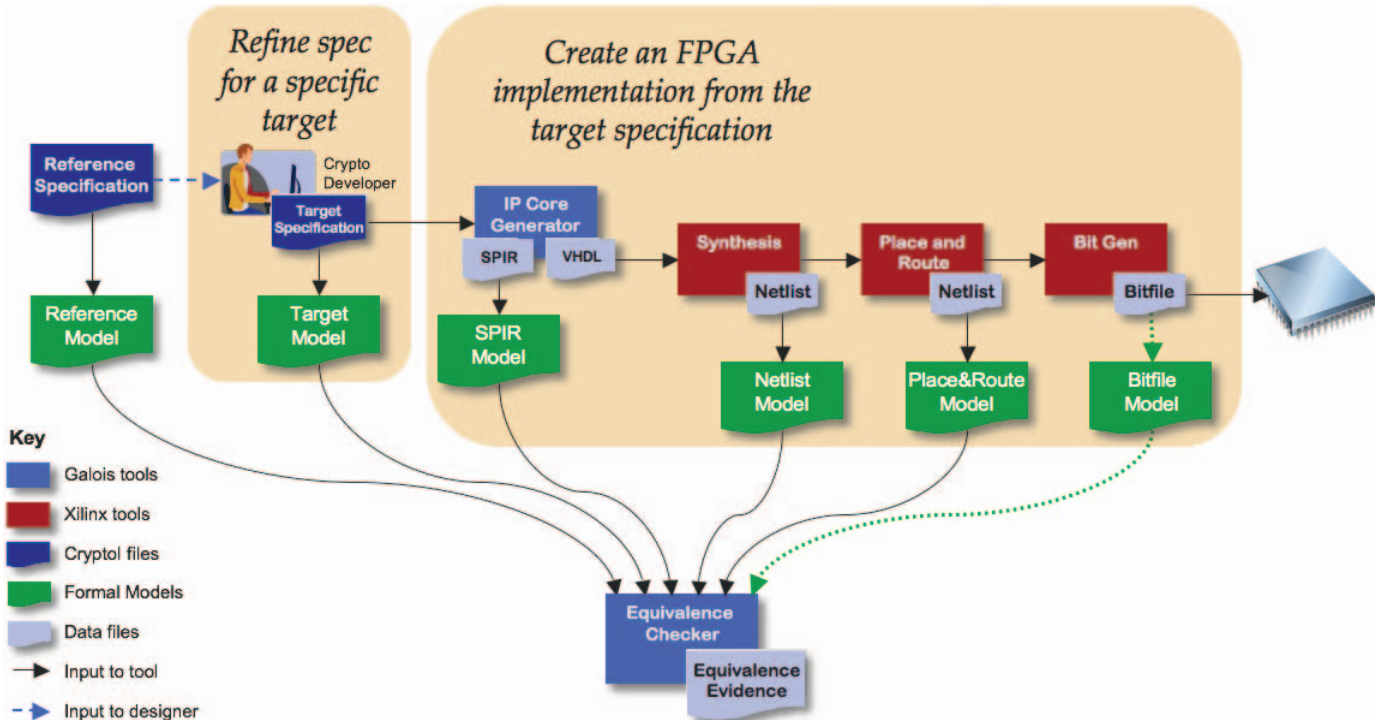


Fig. 1. Typical design and verification flow in Cryptol. Verification can be performed at various points during the translation, which allows for high-assurance refinement during development. Note that the major compiler phases (the flow through the top-line) remain out of the trusted-code base for verification: One only needs to trust the down-arrows representing translators from various intermediate forms to AIG-based formal models, along with the off-the-shelf equivalence checkers themselves.

from state-explosion, and thus might fail to provide an answer in a feasible amount of time for larger designs. Cryptol uses off-the-shelf SAT/SMT solvers (such as ABC or Yices) as the underlying equivalence checking engine, translating Cryptol specifications to appropriate inputs for these tools automatically [4], [5]. However, the use of these external tools remains transparent to the users, who only interact with Cryptol as the main verification tool.

It is essential to emphasize that equivalence checking applies not only to hand-written programs but also to generated code as well. Cryptol's synthesis tools perform extensive and often very complicated transformations to turn Cryptol programs into hardware primitives available on target FPGA platforms. The formal verification framework of Cryptol allows equivalence checking between Cryptol and netlist representations that are generated by various parts of the compiler, as we will explain shortly. Therefore, any potential bugs in the compiler itself are also caught by the same verification framework. This is a crucial aspect of the system: proving the Cryptol compiler correct would be a prohibitively expensive task, if not impossible. Instead, Cryptol provides a *verifying* compiler, generating code together with a formal proof that the output is functionally equivalent to the input program [6], [7]. As opposed to using a deep-embedding in a theorem prover like ACL2, however, we utilize modern off-the-shelf SAT and SMT solvers to perform automated equivalence checking.

A. Design and Verification Flow

Figure 1 provides a high-level overview of a typical Cryptol development and co-verification flow. Starting with a reference specification in Cryptol, the designer successively refines his program and "runs" his design at the Cryptol command line. These refinements typically include various pipelining and structural transformations to increase speed and/or reduce space usage. Behind the scenes, the Cryptol tool-chain translates Cryptol to a custom signal-processing intermediate representation (SPIR), which acts as a bridge between Cryptol and FPGA-based target platforms. The SPIR representation allows for easy experimentation with high-level design changes since it remains fully executable, also providing essential timing/space usage statistics without going through the computationally expensive synthesis tasks.

Once the programmer is happy with the design, Cryptol translates the code to VHDL, which is further fed to third party synthesis tools. Figure 1 shows the flow for the Xilinx tool-chain; taking the VHDL through synthesis, place and route, and bit-file generation steps. In practice, these steps might need to be repeated depending on feedback from the synthesis tools. The overall approach aims at reducing the number of such repetitions greatly, by providing early feedback to the user at the SPIR level. The final outcome is a custom binary file that can be downloaded onto an Xilinx FPGA board, completing the design process.

Our co-verification flow is interleaved with the design process. As depicted in Figure 1, Cryptol provides custom translators at various points in the translation process to generate formal models in terms of AIG (and-inverter graph) representations [8]. In particular, the user can generate AIG representations from the reference (unoptimized) Cryptol spec-

ification, from the target (optimized) Cryptol specification, from the SPIR representation, from the post-synthesis circuit description, and from the final (post-place-and-route) circuit description. By successive equivalence checking of the formal models generated at these check points, Cryptol provides the user with a high-assurance development environment, ensuring that the transformations applied preserve semantic equivalence. The final piece of the puzzle for end-to-end verification is generating an AIG for the bit-file generated by the Xilinx tools, as represented by the dashed line in Figure 1. The format of this file remains proprietary, but we hope to provide this final link through future collaboration with Xilinx.

B. Verification for the Cryptography domain

Cryptol's formal verification framework clearly benefits from recent advances in SAT/SMT solving. However, it is also important to recognize that the properties of cryptographic algorithms make applications of automated formal methods particularly successful. This is especially true for symmetric key encryption algorithms that rely heavily on low-level bit manipulations instead of the high-level mathematical functions employed by public-key cryptography.

In particular, symmetric key crypto-algorithms almost never perform control flow based on input data in order to avoid attacks based on timing. The series of operations performed are typically "fixed," without any dependence on the actual input values. Similarly, the loops used in these algorithms almost always have fixed bounds; typically these bounds arise from the number of rounds specified by the underlying algorithm. Techniques like SAT-sweeping [9] are especially effective on crypto-algorithm verification, since simulation-based node-equivalence guesses are likely to be quite accurate for algorithms that heavily rely on shuffling input bits. Obviously, these properties do not make formal verification trivial for this class of crypto-algorithms; but rather they make the use of such techniques quite feasible in practice [10].

III. Case study: Skein

Skein [11] is a suite of cryptographic hash algorithms targeted at the NIST competition for choosing the next-generation hash function SHA-3 [12]. At its core, Skein uses a tweakable block cipher named Threefish. The unique block iteration (UBI) chaining mode defines the mode of operation by the repeated application of the block cipher function. A detailed write-up on the Cryptol implementation of Skein is publicly available [13].

The process of proving that a given VHDL implementation is functionally equivalent to a Cryptol reference specification begins with understanding the high-level interfaces of both. Once the high-level input/output correspondences are determined, the VHDL implementation is imported into the Cryptol program using Cryptol's `extern` declaration capability. Then, the required interface-matching code is written in Cryptol, mainly taking care of the proper use of control-signals. This allows the external implementation to be available at the Cryptol command prompt, enabling the user to call it on specific values, pass it through previously generated test vectors, essentially making the external definition behave just like any other Cryptol function. This facility greatly increases productivity, since it unifies software and hardware under one common interface. Once the reference specification and the Cryptol/VHDL hybrid expose the same interface, the user generates AIGs from both, and checks for equivalence.

We verified our implementation of Skein against two separate VHDL implementations using this methodology. In each case, we have used Alan Mishchenko's ABC tool as the underlying equivalence checker [4]. The verification was performed for one 256-bit input block, generating a 256-bit hash value.

The first verification was performed against Men Long's implementation [14]. Since Long did not implement the full Skein algorithm, but instead implemented only the underlying Threefish encryption and the XOR of input data, we tweaked our reference Cryptol implementation to match this partial result. The AIG generated for the reference implementation in Cryptol had 118156 and-gates, while the VHDL version gave rise to 653963 and-gates; about 5.5 times larger. The equivalence checking process took about an hour to complete on commodity hardware using ABC. During the verification effort, we encountered a problem with a piece of code that rotates a 64-bit signal a variable number of steps. It was given three different meanings by GHDL [15], simili [16] and the Xilinx synthesis tools. We were able to remove the ambiguity of the code by replacing it with the standard library function `rotate_left`. In essence, the Cryptol co-verification path found an ambiguity bug that remained undetected before.

We performed our second verification against Stefan Tillich's Skein implementation [17], which is a full implementation of the Skein algorithm. The AIG sizes in this case were 301085 and-gates for the reference Cryptol versus 900239 and-gates for the VHDL implementation; about 3 times larger. In this case, equivalence checking was completed in about 18 hours using ABC.

Commonly, the VHDL implementations include a global reset signal that brings back the circuit's internal registers to their initial state. In some cases, the reset signal is acted upon immediately regardless of the clock signal, that is, in an asynchronous manner. Our tools are currently not able to generate formal models from VHDL code with asynchronous resets—instead we have an additional assumption that the reset signal is only asserted initially and in a synchronous manner across a rising clock edge. Under this assumption, we manually perform an equivalence-preserving rewrite of the VHDL code so that it uses the reset signal synchronously. In the case of Tillich's code, this was a local rewrite consisting of a couple of lines of VHDL code change. While this is a problem for any asynchronous signal, we plan to enhance our tools so that they can at least handle asynchronous resets without the need for manual VHDL code modifications.

IV. Challenges

Increasing the coverage of formal methods: Cryptol's formal verification framework works on a (relatively large) subset of Cryptol [3]. The main limitation is in the verification of algorithms for all-time, i.e., those programs that receive

and produce infinite streams of data. While infinite streams pose no challenge for synthesis, Cryptol can only equivalence check such algorithms up to a fixed number of clock-cycles. Although this restriction is irrelevant for most block-based crypto-algorithms, it does not generalize to stream-ciphers in general. The introduction of induction capabilities in the equivalence checker or the use of hybrid methods combining manual top-level proofs with fully-automated SAT/SMT based sub-proofs might provide a feasible alternative for handling such problems.

Proving security properties: Unsurprisingly, not all properties of interest can be cast as functional equivalence problems. This is especially true in the domain of cryptography. For instance, if we are handed an alleged VHDL implementation of AES, we would like to ensure that it not only implements AES correctly but also that it does not contain any extra circuitry to leak the key.

The trusted code base: Cryptol's formal verification system relies on the correctness of the Cryptol compiler's front-end components (i.e., the parser, the type system, etc.), the symbolic simulator, and the translators to SAT/SMT solvers. Note that Cryptol's internal compiler passes, optimizations, and code generators (i.e., the typical compiler back-end components) are *not* in the trusted code base. While Cryptol's trusted code base is only a fraction of the entire Cryptol tool suite, it is nevertheless a large chunk of Haskell code. Reducing the footprint of this trusted code base, and/or increasing assurance in these components of the system is an on-going challenge.

V. Related Work

Software/Hardware co-verification problem is an open research area, especially focusing on equivalence checking C-style programs against RTL implementations [18], [19]. Similar to earlier pioneering work in this area, our approach does not freeze the—usually very large and complicated—code generator portion of the system, since the code generator is not in the trusted path. The reduction of the trusted code base is a significant gain for assurance purposes. Unlike earlier work, however, we do not assume that the implementations in these languages are "similar," or done in certain styles to enable effective verification. In fact, Cryptol specifications remain purely functional and hence combinatorial, while VHDL implementations are typically highly state-based and thus sequential. Note that, our approach remains bit-precise, i.e., no simplifying assumptions are made on the semantics of the underlying languages.

VI. Conclusions

In this paper, we have provided a brief description of the Cryptol language, an overview of its verification framework, and a case study of verifying the hash algorithm Skein. The novel part of our approach is the bridging of hardware and software artifacts, allowing designers to treat them uniformly during verification. The system has already proved itself useful in practice, especially in establishing the equivalence of reference and extensively optimized implementations of crypto-algorithms such as AES. Since such transformations are done both manually by programmers and automatically by the compiler and external synthesis tools, it is essential that automated formal-verification capabilities are seamlessly integrated into the process, ensuring that the final hardware implementations are semantically equivalent to their high-level reference specifications.

Acknowledgments

Many people have worked on Cryptol and its formal verification toolset over the years, including Jeff Lewis, Thomas Nordin, John Matthews, Sigbjorn Finne, Phil Weaver, and Sally Browning. Men Long of Intel and Stefan Tillich of TU Graz kindly made their VHDL code available to us for verification and answered several questions on their implementations.

References

[1] J. R. Lewis and B. Martin, "Cryptol: high assurance, retargetable crypto development and validation," in *Military Communications Conference 2003*, vol. 2. IEEE, Oct. 2003, pp. 820–825.

[2] NIST, "Announcing the AES," November 2001, FIPS Publication 197. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] L. Erkök and J. Matthews, "Pragmatic equivalence and safety checking in Cryptol," in *Programming Languages meets Program Verification, PLPV'09, Savannah, Georgia, USA*. ACM Press, Jan. 2009, pp. 73–81.

[4] A. Mishchenko, "ABC: System for sequential synthesis and verification," 2007, release 70930, Available at: http://www.eecs.berkeley.edu/~alanmi/abc.

[5] "Yices web site," http://yices.csl.sri.com/.

[6] L. Pike, M. Shields, and J. Matthews, "A verifying core for a cryptographic language compiler," in *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. New York, NY, USA: ACM, 2006, pp. 1–10.

[7] W. A. Hunt and E. Reeber, "Formalization of the DE2 language," in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3725. Springer, 2005, pp. 20–34.

[8] A. Biere, "The AIGER And-Inverter Graph (AIG) format," 2007. [Online]. Available: http://fmv.jku.at/aiger/FORMAT.aiger

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002.

[10] E. W. Smith and D. L. Dill, "Automatic formal verification of block cipher implementations," in *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–7.

[11] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The Skein Hash Function Family," 2009, http://www.skein-hash.info.

[12] "NIST Cryptographic Hash Algorithm Competition," 2008, http://csrc.nist.gov/groups/ST/hash/sha-3.

[13] S. Finne, "A Cryptol implementation of Skein," 2009, http://www.galois.com/blog/2009/01/23/a-cryptol-implementation-of-skein.

[14] M. Long, "Implementing Skein hash function on Xilinx Virtex-5 FPGA platform," 2009, http://www.skein-hash.info/sites/default/files/skein_fpga.pdf.

[15] "GHDL simulator version 0.26," http://ghdl.free.fr/.

[16] "Simili VHDL simulator version 3.1," http://www.symphonyeda.com/products.htm.

[17] S. Tillich, 2009, The Institute for Applied Information Processing and Communications (IAIK). http://www.iaik.tugraz.at/content/research.

[18] A. Pnueli, O. Shtrichman, and M. Siegel, "The code validation tool (cvt) - automatic verification of code generated from synchronous languages," *Software Tools for Technology Transfer*, vol. 2, 1998.

[19] L. Séméria, R. Mehra, B. M. Pangrle, A. Ekanayake, A. Seawright, and D. Ng, "Rtl c-based methodology for designing and verifying a multi-threaded processor," in *Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002*, 2002, pp. 123–128.

Retiming and Resynthesis with Sweep Are Complete for Sequential Transformation

Hai Zhou
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208

Abstract— **There is a long history of investigations and debates on whether a sequence of retiming and resynthesis is complete for all sequential transformations (on steady states). It has been shown that the sweep operation, which adds or removes registers not used by any output, is necessary for some sequential transformations. However, it is an open question whether retiming and resynthesis with sweep are complete. This paper proves that the operations are complete, but with one caveat: at least one resynthesis operation needs to look through the register boundary into the logic of previous cycle. We showed that this one-cycle reachability is required for retiming and resynthesis to be complete for re-encodings with different code length. This requirement comes from the fact that Boolean circuit is used for a discrete function thus its range needs to be computed by a traversal of the circuit. In theory, five operations in the order of sweep, resynthesis, retiming, resynthesis, and sweep are already complete. However, some practical limitations on resynthesis must be considered. The complexity of retiming and resynthesis verification is also discussed.**

I. Introduction

Logic synthesis algorithms originally targeted the optimization of PLA implementations; this was followed by research in synthesizing more general multilevel logic implementations. Currently, the central thrust in logic synthesis is sequential synthesis, i.e., the automatic optimization of the entire system. This is for designs specified at the structural level in the form of netlists, or at the behavioral level, i.e., in the form of finite state machines. DeMicheli [12] gives an excellent introduction to logic synthesis.

In this paper, we will be concerned with sequential designs. These can be specified at the behavioral level, as *finite state machines* (FSMs), or at the structural level, as *netlists* of *gates* and *registers*.

Retiming is a powerful sequential optimization step that can be applied to sequential designs described at the netlist level. It can be used to optimize the clock period or the registers area of a design. Logic synthesis is an operation that changes the circuit structure without changing the function of the combinational logic. It has been shown that given two designs, one of netlists has been derived from the other by a sequence of retiming and resynthesis, a certain equivalence relation (namely, steady-state equivalence) exists between them. However, the converse is not well understood, and there is a long history of investigations and debates on whether a sequence of retiming and resynthesis is complete for any sequentially equivalent transformation.

Malik [11] gave the first (partial) positive answer to this question. He proved that retiming and resynthesis are complete for any state re-encoding, and for some other transformations. Zhou et al. [18] provided the first negative answer by proving that some sequentially equivalent transformations cannot be done by retiming and resynthesis, which also helped to discover and fix an error in Malik's result [13]. The sweep operation, which adds or removes registers not used by any output, is needed for these transformations. However, it is an open question whether retiming and resynthesis with sweep are complete for general sequential transformations.

In this paper, we provide a complete answer to the open question. We proved that retiming and resynthesis with sweep are complete, but with one caveat: at least one resynthesis operation needs to look through the register boundary into the logic of previous cycle. We even showed that this one-cycle reachability is required for retiming and resynthesis to be complete for re-encodings with different code length, an extension to Malik et al. [10]. It also demonstrates that reachability information cannot be captured by these structural operations. Therefore, they are complete for transformations based on all steady states unless reachability information is provided. Our completeness proof is a constructive one that applies five operations in the order of sweep, resynthesis, retiming, resynthesis, and sweep. We will discuss the implications of such a result and some practical limitations on resynthesis.

Zhou et al. [18] also started an investigation on the complexity of retiming and resynthesis verification problem. Since the general sequential equivalence verification is PSPACE-complete, a different complexity category may indicate that the gap between retiming and resynthesis and sequential transformation is big. Jiang and Brayton [6] later showed that the complexity of retiming and resynthesis verification is also PSPACE-complete. We examine their proof and point out parts that are unclear. Based on those we consider the membership of retiming and resynthesis verification an open question.

Our results have very important practical implications. Since retiming and resynthesis with sweep are complete, sequential optimization tools can be centered around them. If any reachability information is provided to the optimization, it is also critical to be supplied to the verification. Our completeness proof also indicates that the resynthesis needs to generate exponential-size circuits to complete some transformations

This work is supported by the NSF under CNS-0613967 and CCR-0811270.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

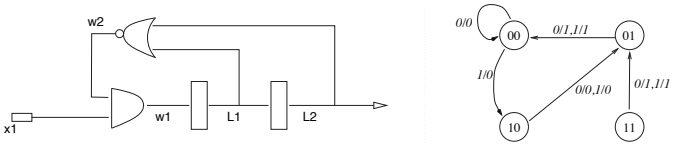


Fig. 1. Netlist and corresponding FSM.

(including some re-encoding ones). However, no practical resynthesis is so powerful. Under realistic limitations, retiming and resynthesis verification is much simpler. Indeed, the recent sequential equivalence checking algorithms [15], [17], [7] effectively try to show that two circuits are equivalent by deriving the retiming relationships between them.

II. Background

Netlists and FSMs: We introduce two formalisms for representing designs, namely netlists and finite state machines (FSMs). Netlists are structural and consist of an interconnection of gates and registers. Finite state machines are behavioral and specify how the system changes its states and produces outputs responding to inputs. We leave for the readers to ponder which representation is more abstract.

Before giving formal definitions to netlists and FSMs, we illustrate them by means of examples. The netlist in Figure 1 has one primary input, one primary output, two registers, and two gates. The FSM for the same design is shown beside it; it consists of 4 states.

Definition 1: A *Finite State Machine (FSM)* is a quintuple $(Q, I, O, \lambda, \delta)$ where Q is a finite set referred to as the *states*, I, and O are finite sets referred to as the set of *inputs* and *outputs* respectively, $\delta : Q \times I \rightarrow Q$ is the *next-state function*, and $\lambda : Q \times I \rightarrow O$ is the *output function.*

The output and next state functions can be inductively extended to the domains $Q \times I^+ \rightarrow O^+$ and $Q \times I^+ \rightarrow Q$, respectively; we continue to use λ and δ to denote these extended functions. For example, for the FSM in Figure 1, $\lambda(10, 1 \cdot 0) = 0 \cdot 1$ and $\delta(01, 0 \cdot 1 \cdot 0) = 01$.

Definition 2: A *netlist* is a directed graph, where the nodes correspond to elementary circuit elements, and the edges correspond to wires connecting these elements. Each node is labeled with a distinct variable w_i. For simplicity, we will assume that the netlist is *Boolean*, i.e. all variables take values in $B = \{0, 1\}$. The three basic circuit elements are *primary inputs, registers*, and *gates*. Primary input nodes have no fanins; registers have a single input. Associated with a gate g on n-inputs $w_1, w_2, \ldots, w_n$ is a function from B^n to B. Some nodes are designated as being *primary outputs*.

Given a value to each input and a state (an assignment of values to registers), one can uniquely compute the value of each node in the netlist by evaluating the functions at gates. A netlist η on inputs $i_1, i_2, \ldots, i_n$, outputs $o_1, o_2, \ldots, o_m$ and registers $r_1, r_2, \ldots, r_k$ bears a natural correspondence to an FSM M_η on inputs $X = B^n$, outputs $Y = B^m$, and state space $Q = B^k$. The next-state function of M_η is defined by the composed logic gates in the following manner: for each register r_i we can find a function $f_i : Q \times X \rightarrow B$ by composing the functions of the gates from the inputs and register outputs to the input of the register. We will refer to f_i as the next-state function of the register i. Then $\delta_{M_\eta} : Q \times X \rightarrow Q$ is simply $(f_1, f_2, \ldots, f_k)$. Similarly, the output function is defined by composing the functions of gates from inputs and registers to output nodes.

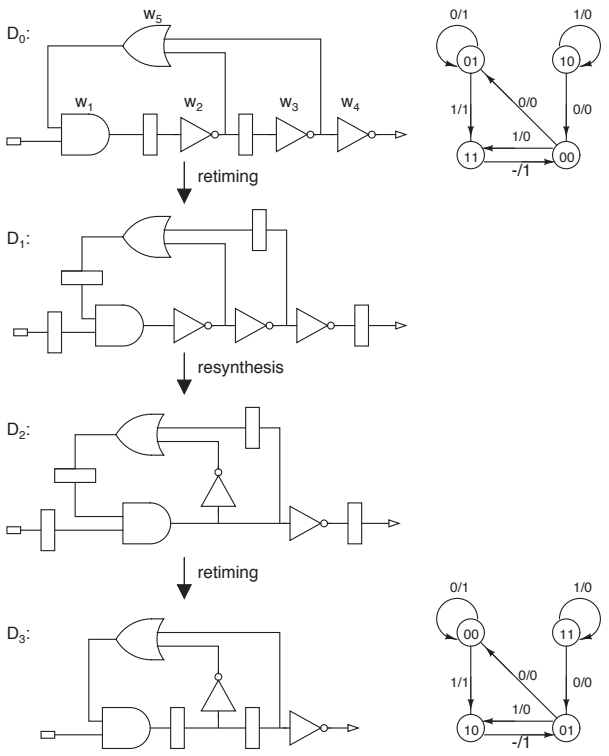


Fig. 2. Example of retiming and resynthesis.

Retiming, Resynthesis, and Sweep: Retiming, resynthesis, and sweep are structural operations applied on netlists.

Retiming consists of moving a given number of registers between the inputs and outputs of each combinational node. A retiming can be described mathematically by a lag function, which gives for each combinational node, the number of registers that are moved from each fanout to each fanin.

Combinational synthesis restructures the netlist within the register boundaries without changing its functionality. It leaves the FSM of the design unchanged. Retiming becomes very powerful when it is interspersed with resynthesis of the netlist within the changed register boundaries. Resynthesis provides new signals for retiming to place registers on while retiming provides new combinational blocks for resynthesis to manipulate. This is the basis for the retiming and resynthesis (RnR) paradigm proposed in [4], [10].

Sweep, the simplest among the operations, adds or removes registers not used by any output. Since synthesis normally simplifies the circuit structure, sweep is usually met as an operation removing redundant registers and logic.

An example of a design transformation using this RnR paradigm is shown in Figure 2. At the first retiming step, we have the following lag function: $r(w_1) = 1, r(w_3) = r(w_4) = -1, r(w_2) = r(w_5) = 0$.

Sequential Equivalence:

Definition 3: Two states s and t are *equivalent*, denoted as $s \cong t$, if and only if for every finite input sequence π, the outputs resulting on applying π are equal.

For example, in circuits D_0 and D_3 of Figure 2, state 00 in D_0 is equivalent to state 01 in D_3.

Definition 4: Two netlists C and D are *FSM-equivalent* if and only if every state $c \in C$ is equivalent to some state $d \in D$, and every state $d \in D$ is equivalent to some state $c \in C$.

Thus the two designs D_0 and D_3 in Figure 2 are FSM-equivalent.

Definition 5: The *steady state set* of a design D, denoted by D^∞ is the subset of states such that for each state s there is an input sequence π which drives this state to itself, i.e., $\lambda_D(s, \pi) = s$. The remaining set of states is called the *transient state set*.

For example, the steady state set for the design in Figure 1 is $\{00, 01, 10\}$ and the transient state set is $\{11\}$. Notice that once a design starts up in any state, it will eventually be and remain in steady states.

Theorem 1 ([8]): If design C has been obtained from design D by a sequence of retiming moves, the steady state set of C is FSM-equivalent to the steady state set of D.

Retiming becomes very powerful when it is combined with (combinational) resynthesis operations (the RnR paradigm). However, since resynthesis itself does not change the state transition graph of a design, we have the following corollary.

Corollary 1: If design C has been obtained from design D by a sequence of retiming and combinational resynthesis moves, the steady state set of C is FSM-equivalent to the steady state set of D.

Designated Initial State: We will not assume a designated initial state for our circuits. If we do want to force a circuits into a designated initial state we can explicitly model the reset circuitry along with the registers: indeed, this is the approach suggested for retiming initial states in [14], as opposed to the approach in [16], [5], where the implicit initial state values have to be retimed across gates.

One optimization advantage of considering designated initial states is that the synthesis algorithms have greater flexibility since the synthesis tool can potentially take advantage of don't cares arising from the set of states unreachable from the initial state. However, it is easy to show that for designs which have designated initial state, retiming and resynthesis is strictly weaker than a sequential optimization algorithm which uses unreachability don't cares (for example, [9]).

Consider circuits C and D in Figure 3 with 00 as the designated initial state for both C and D. Clearly the two circuits C and D are equivalent from the initial state 00. However, from Corollary 1, it is clear that C and D are not RnR equivalent (since state $10 \in C^\infty$, the steady state set of C, but there is no equivalent state in D^∞).

However, in general, commercial synthesis tools do not use unreachability don't cares. This is simply because computing the set of unreachable states is computationally very expensive on real designs; the theoretical complexity of this problem is PSPACE-complete:

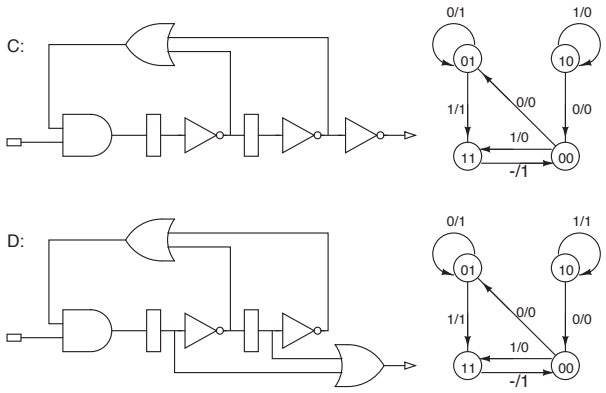


Fig. 3. Circuits C and D are equivalent from designated initial state 00.

Theorem 2 ([2]): Given two netlists C and D, and two states s from M_C, t from M_D, checking whether s and t are equivalent is PSPACE-complete in the size of the netlists.

III. Sweep is Necessary

Even though it is commonly suspected that retiming and resynthesis are not complete for all steady state equivalent transformations, Zhou et al. [18] was the first giving such a proof. They designed two pairs of circuits and proved that the first pair cannot be transformed to each other even though they are FSM-equivalent. The second pair was also conjectured so. We show here that both pairs are incomplete, using the same reasoning they used for the first pair. The two pairs of circuits are shown in Figure 4.

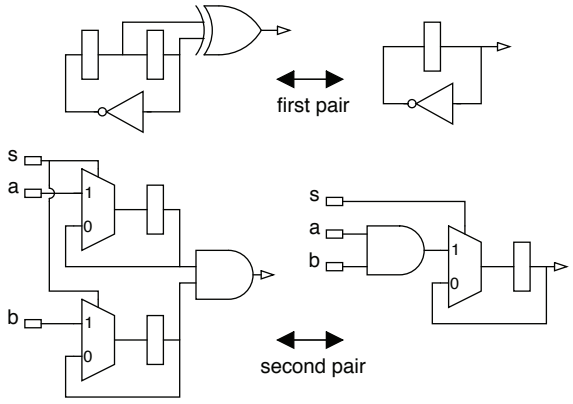


Fig. 4. Examples showing incompleteness of retiming and resynthesis.

Lemma 3: Retiming and resynthesis cannot transform one circuit to the other for either pair in Figure 4.

Proof: The next state function of the left circuit in each pair contains a permutation on the set $\{0, 1\}^2$, which has cardinality of 4. No matter what resynthesis does, the smallest cut size, in terms of the number of signals, on the combinational part must be at least 2, in order to encode all the information. Therefore, the next retiming step cannot reduce the number of registers. Since the next state function of the new circuit still has the property as the old one, any later retiming and resynthesis steps cannot reduce the number of

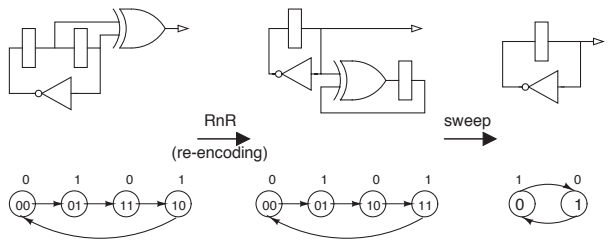


Fig. 5. Retiming and resynthesis are more powerful with sweep.

registers, either. This means that no sequence of retiming and resynthesis can transform the left circuits to the right ones. ■

However, it was also noted in [18] that with the sweep operation, the first pair of circuits are transformable to each other, as shown in Figure 5. We investigate whether the sweep is also of help in the second pair of circuits and find that, with re-encoding and sweep, they can be transformed, as shown in Figure 6. However, when trying to design a sequence of retiming and resynthesis to do the re-encoding, instead of direct applying Malik's theorem, we found that re-encoding is harder than previous thought and resynthesis needs to be slightly enhanced for retiming to be complete for re-encoding. The details will be presented in the next section.

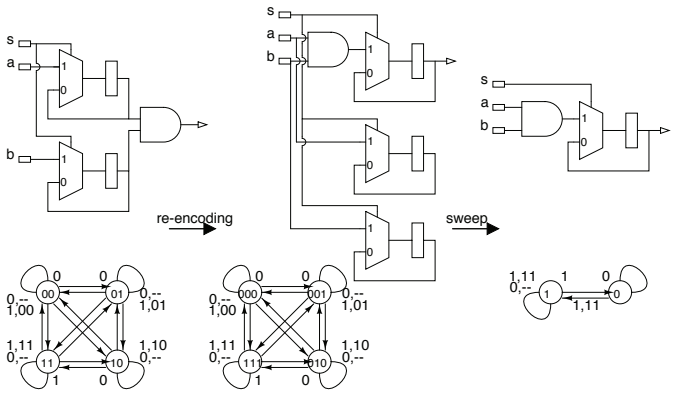


Fig. 6. Second pair is completed by re-encoding and sweep.

IV. Re-encoding is Hard

The first attempt to relate re-encoding and retiming-and-resynthesis was made by Malik et al. [10] via the following result which relates designs with different state encoding:

Theorem 4 ([10]): If two circuits have the same symbolic FSM, then one circuit can be obtained from another by a sequence of retiming and resynthesis.

However, the above theorem cannot be applied to re-encodings with different code length. we have the following result. This result also shows a sharp difference between reachability and retiming-and-resynthesis.

Lemma 5: Without any reachability information, some re-encodings with different code length cannot be completed by any sequence of retiming and resynthesis.

Proof: As we already mentioned in previous section, in Figure 6, even though the second circuit is a re-encoding of the first one, it cannot be transformed from the first one by a sequence of retiming and resynthesis. This can be proved by considering all the states in the second circuit, including all the ignored unreachable states, as shown in Figure 7. Since new cycles are created in this STG of the second circuit, retiming an resynthesis simply cannot produce such a circuit from the first one. This is based on Jiang and Brayton [6]'s characterization of STG transformations by retiming, in which no cycle can be generated or canceled. ■

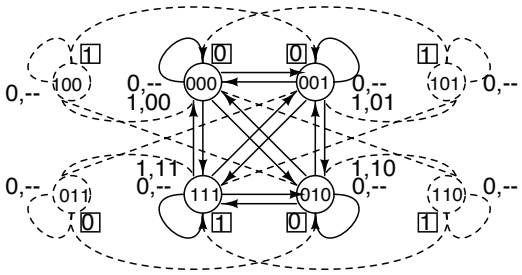


Fig. 7. Unreachable states need to be considered in re-encoding of different length.

Although re-encodings with same code length can be done by retiming and resynthesis, their verification problem is not easy. The following theorem shows that the problem is PSPACE-hard by reducing reachability problem to it.

Theorem 6 ([3]): Checking whether two circuits with the same number of registers are re-encoding of each other is PSPACE-hard.

In [6] an answer about the membership of retiming and resynthesis equivalence in PSPACE is explored. Retiming and resynthesis equivalence is reduced to immediate equivalent state minimization of the two machines and then graph isomorphism starting from known initial states. It is unclear though how graph isomorphism can be checked in PSPACE. Moreover, the proof for completeness is based on the reduction of reachability to checking whether the State Transition Graphs of the two circuits are isomorphic including the transient states. The assumption is that all dangling[1] states can be merged to non-dangling states. However, due to the binary representation of the FSM, this is not always possible. An example can be seen in Figure 8 in which no retiming and resynthesis transformation can merge the immediate equivalent states s_1 and s_3 . The reason is that for n registers the number of states in the State Transition Graph must be 2^n. When binary representation is used and the dangling states cannot be ignored, the State Transition Graphs of two retiming and resynthesis equivalent circuits may not be transformable to isomorphic graphs.

V. Completeness under Reachability

We first show a revised result for re-encoding transformation.

[1]Dangling states are inductively defined as states that have no predecessors or states whose predecessors are all dangling. All other states are considered non-dangling.

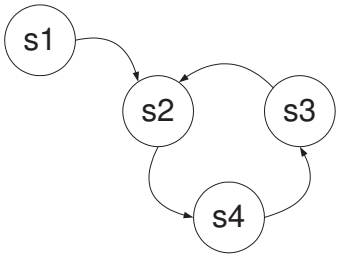


Fig. 8. State transition graph. Immediate equivalent states s_1 and s_3 cannot be merged using retiming and resynthesis in a circuit with binary representation.

Lemma 7: When the resynthesis is allowed to use the reachability information generated from one cycle, retiming and resynthesis are complete for all re-encoding transformations, including those with different coding lengths.

Proof: The proof is similar to Malik et al. [10], using schematics for circuits in Figure 9. Starting with a circuit C with the smaller encoding length n, the identity function at the register outputs is resynthesized to $f \cdot f^{-1}$ where f is the one-to-one mapping from states of C to the target states of circuit D. Please note that f may not be an onto function, thus f^{-1} may be different depending on how to map the don't care states in D. Then retiming moves the registers forward over f. The third step resynthesizes $f^{-1} \cdot C \cdot f$ into D. However, if D is encoded on a longer length m, the one-cycle reachability information needs to be used to identify the target states corresponding to states in C, which has to be used for generating D in the last step. ■

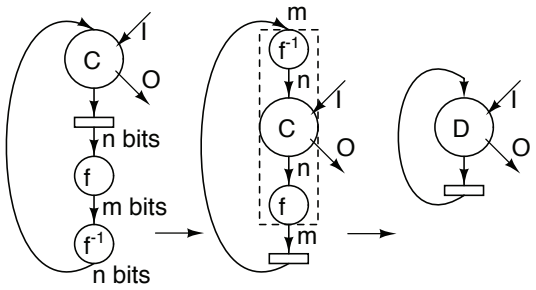


Fig. 9. One-cycle reachability makes retiming and resynthesis complete for re-encodings.

The key to the completeness of retiming and resynthesis for re-encodings is the existence of a mapping from the states of one machine to those of the other that preserves the transitions. Such a mapping is called *refinement mapping* [1].

Definition 6: For two equivalent finite state machines $(Q_1, I, O, \lambda_1, \delta_1)$ and $(Q_2, I, O, \lambda_2, \delta_2)$, a *refinement mapping* is a function $f : Q_1 \to Q_2$ such that for any $s \in Q_1$, s and $f(s)$ are equivalent, and further for any $i \in I$,

$$f(\delta(s, i)) = \delta(f(s), i).$$

Abadi and Lamport [1], studying the verification of one system implementing another, proved that, if S_1 implements S_2, then one can add auxiliary history and prophecy variables to S_1 to form an equivalent system S_1^{hp} and find a refinement mapping from S_1^{hp} to S_2 under three very general hypotheses: S_1 is machine closed, S_2 has finite invisible nondeterminism, and S_2 is internally continuous. For deterministic finite state machines, they are always true. The following result is simply a corollary of the main theorem of Abadi and Lamport [1]. But we will give a direct proof to avoid detouring via general (infinite nondeterministic) system models.

Theorem 8: If two deterministic FSMs C and D are equivalent, then one can add history variables to C to form an equivalent FSM C', and find an onto refinement mapping from C' to D.

Proof: For $C = (Q_C, I, O, \lambda_C, \delta_C)$ and $D = (Q_D, I, O, \lambda_D, \delta_D)$, we can have

$$\begin{aligned} Q_{C'} &\triangleq \{(c, d) \in Q_C \times Q_D : c \cong d\} \\ \lambda_{C'}((c, d), i) &\triangleq \lambda_C(c, i) \\ \delta_{C'}((c, d), i) &\triangleq (\delta_C(c, i), \delta_D(d, i)) \end{aligned}$$

It is straight-forward to check that $f(c, d) = d$ for any $(c, d) \in Q_{C'}$ is a refinement mapping from $C' \triangleq (Q_{C'}, I, O, \lambda_{C'}, \delta_{C'})$ to D. ■

The proof only gives a simple construction without considering efficiency; for any states c and d such that $c \cong d$, we need only add a history variable to record the part of d that is independent of c, instead of the whole d. In the special case where each d is totally dependent on c, no history variable is needed, and the refinement mapping is the generating function of d from c.

With the refinement mapping, a completeness result can be given as follows.

Theorem 9: If two circuits are equivalent, then one of them can be transformed to the other by a sequence of sweep (inverse), resynthesis, retiming, resynthesis, and sweep, given that the second resynthesis operation is allowed to use one-cycle reachability.

Proof: For two equivalent circuits C and D, their corresponding FSMs are deterministic and equivalent. Based on Theorem 8, a set of history registers and their next state functions can be added to C to make it C', and an onto refinement mapping can be found from C' to D. Denote the mapping by F. Adding unobservable registers and their next state functions is just an inverse of the sweep operation.

If F is an one-to-one mapping, then F^{-1} exists. Otherwise, we expand F with the register outputs of C and denote by F^{-1} the function that generate the state of C' from the output of F. Resynthesis can generate F and F^{-1} connected at the register output of circuit C'. Then retiming moves the registers to the outputs of F. Since F is a refinement mapping from C' to D, the relocated registers give the states of circuit D in parallel with (possibly partial) states of circuit C. The circuit composed of F^{-1}, H (the history transition), and F can be re-synthesized into the circuit D in parallel with another circuit (partial C). Then a sweep operation will remove all unobservable part to produce circuit D. The sequence of the five operations are shown in Figure 10. A key operation in the second re-synthesis operation is to have the output from D, instead of C. This cannot be done if the register vectors V_c and V_d are assumed to be independent (as in pure combinational

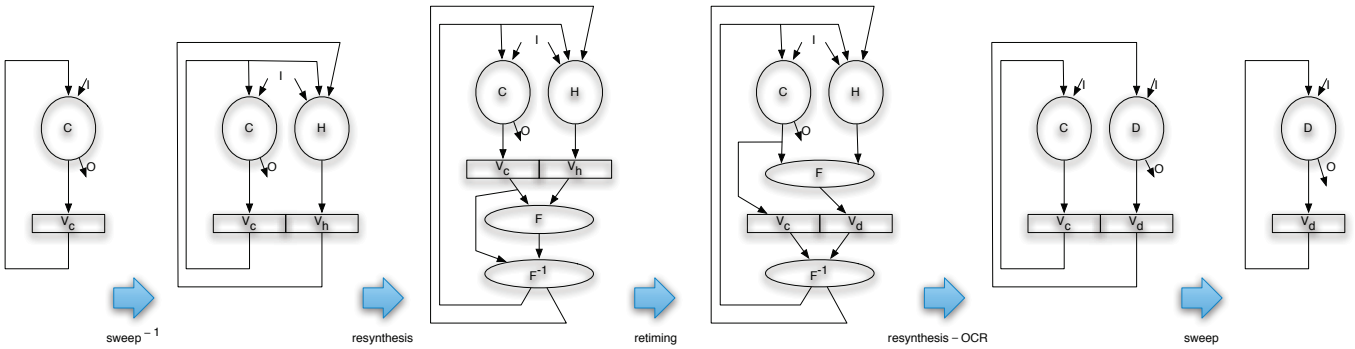


Fig. 10. Transformation from a circuit to an equivalent one by retiming and resynthesis with sweep.

synthesis). However, with the observation that $V_d = F(V_c)$ from the previous cycle, the output O can be synthesized out solely from V_d. ■

VI. Conclusions

We have shown in this paper that retiming and resynthesis with sweep are almost complete for all steady state equivalent transformations, in the sense that resynthesis needs to get one-cycle reachability information by looking into previous phase. Without such information, they cannot even complete re-encodings with different code length. It suggests that a powerful sequential optimization tool can be built around retiming, resynthesis, and sweep, and also suggests to enhance each resynthesis step to employ one-cycle reachability by looking into previous phase. In practice, resynthesis may not generate exponential-size circuits and may have other restrictions. Those restrictions can make the retiming and resynthesis equivalence checking easier.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2), 1991.

[2] A. Aziz, V. Singhal, and R. K. Brayton. Verifying Interacting Finite State Machines. Technical Report UCB/ERL M93/52, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.

[3] José L. Balcázar, Antoni Lozano, and Jacobo Torán. The complexity of algorithmic problems on succinct instances. *Computer science: research and applications*, pages 351–377, 1992.

[4] G. DeMicheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE TCAD*, 10(1):63–73, January 1991.

[5] G. Even, I. Y. Spillinger, and L. Stok. Retiming Revisited and Reversed. *IEEE TCAD*, 15(3):348–357, March 1996.

[6] J.-H. Jiang and R. Brayton. Retiming and resynthesis: A complexity perspective. *IEEE TCAD*, 25:2674–2686, December 2006.

[7] J.-H. Jiang and W.-L. Hung. Inductive equivalence checking under retiming and resynthesis. In *ICCAD*, San Jose, CA, November 2007.

[8] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[9] B. Lin, H. J. Touati, and A. R. Newton. Don't Care Minimization of Multi-level Sequential Logic Networks. In *ICCAD*, pages 414–417, November 1990.

[10] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques. *IEEE TCAD*, 10(1):74–84, January 1991.

[11] Sharad Malik. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, November 1990. Memorandum No. UCB/ERL M90/115.

[12] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[13] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton. On the optimization power of retiming and resynthesis transformations. In *ICCAD*, pages 402–407, San Jose, CA, November 1998.

[14] V. Singhal, S. Malik, and R. K. Brayton. The Case for Retiming with Explicit Reset Circuitry. In *ICCAD*, pages 618–625, November 1996.

[15] D. Stoffel and W. Kunz. Record and Play: a Structural Fixed Point Iteration for Sequential Circuit Verification. In *ICCAD*, pages 394–399, 1997.

[16] H. J. Touati and R. K. Brayton. Computing the Initial States of Retimed Circuits. *IEEE TCAD*, 12(1):157–162, January 1993.

[17] C. A. J. van Eijk. Sequential Equivalence Checking without State Space Traversal . In *DATE*, pages 618–623, Paris, France, 1998.

[18] H. Zhou, V. Singhal, and A. Aziz. How powerful is retiming? In *Workshop Notes of Intl. Workshop on Logic Synthesis*, Lake Tahoe, CA, June 1998.

SAT-based Synthesis of Clock Gating Functions Using 3-Valued Abstraction

Eli Arbel, Oleg Rokhlenko, and Karen Yorav
IBM Haifa Research Laboratory
e-mail:{arbel,olegr,yorav}@il.ibm.com

***Abstract*—Clock gating is a power reduction technique for digital circuits that works by eliminating unnecessary switching of parts of the clock network, a power-hungry component in hardware designs. An effective approach to clock gating synthesis is based on a functional analysis of the design using BDDs. Algorithms of this type attempt to build a BDD for a clock gating circuit and then reduce its size with an approximation. If the BDD of a particular latch grows too large the attempt to gate that latch is aborted.**

We replace BDDs with a SAT-based technique combined with 3-valued abstraction. Our technique generates the approximation directly from the circuit, and thus avoids the explosion. Furthermore, our technique is incremental in the sense that it produces a partial result (a weaker approximation) if time or memory limits are exceeded. Our experimentation shows that more than 70% of latches that could not be gated using the BDD-based approach were gated by the SAT-based method.

I. Introduction

Power consumption of digital systems has become just as important as performance. VLSI designers spend significant amounts of time optimizing their designs for minimal power consumption, using many different techniques. Current research is challenged to automate and optimize power reduction techniques so that they can match, and outperform, the efficiency of hand-crafted optimizations.

Clock gating is a power reduction technique that is used at the netlist (or synthesis) level. In clock gating the clock input of a latch (in this paper a latch signifies any type of memory element) is prevented from "ticking" if a tick would be redundant. This can happen, for example, when it is guaranteed that the output of the latch is not about to change, or when the resulting computation is not going to be used. Clock gating is achieved by computing a *gating function*, which is a Boolean function that evaluates to '1' when the clock should be gated (i.e. prevented from ticking).

Our focus is on clock gating synthesis algorithms based on functional analysis (as opposed to structural methods). These algorithms can be roughly described by a general flow that includes: (1) constructing a Boolean circuit C that represents clock gating opportunities for a given latch (2) building a BDD for C (3) synthesizing an optimized clock gating function out of the BDD. Different clock gating algorithms use different circuit constructions. We use a construction called "data-independent feedback loop elimination" as an example of an algorithm for the construction of C. However, our method is applicable to all algorithms that fit the above described flow.

The BDD-based flow is efficient in most cases and can significantly reduce the power consumption of a design. However, since the circuit C is built based on the next-state function of the corresponding latch, usually including more than one copy of this function, the method works well only when the next-state function of the latch is relatively small. The common practice is that when the BDD size exceeds some limit the attempt to gate that latch is aborted. For some designs, especially large ones, this leaves significant parts of the design with no clock gating even though in theory these elements can be efficiently gated.

Our motivation is to replace BDDs with a SAT-based method, in those cases where BDDs are not useful. We would like to retain the ability to generate the strongest clock gating function, while avoiding memory explosion. BDD-based methods attempt to build a BDD for the entire clock gating circuit, and then reduce its size with an approximation. In contrast, our SAT-based technique can generate the same approximation directly from the circuit, and thus avoid the explosion.

Our method is based on a satisfiability check of a formula that is derived from the circuit constructed by the clock gating algorithm. Every satisfying assignment to this formula represents a clock gating opportunity. However, what we need are bounded-size terms, which are generated from full assignments by universal quantification over input signals. Inherently, SAT solvers compute existential quantification and computing universal quantification iteratively would be too expensive. The core of our proposed method is to use X-values to perform (an approximation of) universal quantification using a SAT-solver. If there is a satisfying assignment in which some input signal is assigned with X it implies that this input can be universally quantified out of the formula (in the context of this assignment). We use this insight, together with a cardinality constraint on the number of inputs allowed to take on X-values, to produce terms in a clock gating function. The constraint is implemented through the use of auxiliary variables that control which inputs take on X-values.

Given enough time our SAT-based method is able to generate the same (strongest) approximation that the BDD-based method produces. Furthermore, even if the SAT-based method does not complete its computation within a set time limit, it will produce a partial result, which is a weaker approximation. We employ the SAT-based method for latches that can not be handled by BDDs. Our experimentation shows that more than 70% of latches that could not be gated using the BDD-based

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

approach were gated by the SAT-based method.

A. Related work

Several approaches for synthesis of clock gating conditions have been explored in the literature. Structural analysis methods [9], [4] identify opportunities based on pre-defined structures or logic gates in the design. By avoiding functional analysis these methods are usually scalable enough to handle large designs. Some methods produce candidates for clock gating by examining simulation traces [24], [12]. Since simulation can provide only a partial information on the behavior of the design, these candidates still need to be proven for correctness in order to be considered valid. For example, a SAT-solver is used in [12] for such task. In contrast, our method produces solutions that are correct-by-construction.

Functional analysis algorithms based on symbolic methods [1], [23], [5], [6] also exist. While usually powerful, being based on BDDs, these methods may not scale well to large designs or may not handle designs with complex logic.

Our clock gating synthesis algorithm uses a satisfiability solver to generate solutions to a problem by adding control variables. Variants of this style of exploiting satisfiable instances have been previously used for bi-decomposing Boolean functions [15] and fault localization [21], [10]. In most applications these extra variables are used to describe the specific solution that is discovered by the solver, while we use them to gauge the amount of abstraction by controlling the number of injected X-values.

Three-valued logic is often used in formal verification in order to create preserving abstractions, for example in bounded model-checking [19] or symbolic trajectory evaluation (STE) [11], [17]. A method for refining STE abstractions using SAT, auxiliary control variables and three-valued logic is described in [18].

While the use of SAT-solvers has been suggested in works on logic synthesis [16], [25] and leakage power reduction [2], to the best of our knowledge ours is the first SAT-based clock gating synthesis algorithm.

II. Background

A *combinational circuit* is a circuit that includes only gates (AND / OR / NOT / XOR / etc.) and no latches. Any combinational circuit can be described by a propositional formula (or a collection of formulas if there is more than one output) such that each input signal becomes an input variable and each gate becomes a logical operator. We use circuits and formulas interchangeably.

In 3-valued logic we use the value X, which stands for "unknown", to abstract away from the specific Boolean values $\{0, 1\}$. The truth tables of propositional operators are extended to the ternary domain $\{0, 1, X\}$ as defined in Figure 1. Evaluating combinational circuits over the ternary domain is a "safe abstraction" in the sense that if the circuit evaluates to 0 (or 1) then it would evaluate to the same value for any assignment to the inputs in which X-values are replaced by Boolean values. Formally, let $f(I)$ be a propositional formula of some circuit

$\wedge$	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

$\vee$	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

$\neg$	
0	1
1	0
X	X

Fig. 1. 3-valued interpretation of basic propositional operators

with inputs I and output o, let $\sigma : I \rightarrow \{0, 1, X\}$ be a 3-valued assignment to the inputs, and $b \in \{0, 1\}$ a Boolean constant. Then for any $i \in I$ such that $\sigma(i) = X$ it holds that $(f(\sigma) = b) \rightarrow (\forall i. f(\sigma) = b)$.

A SAT-solver is a tool that takes a Boolean formula $f(V)$ over a set of variables V and finds an assignment $\sigma : V \rightarrow \{0, 1\}$ s.t. $f(\sigma) = 1$. The input to the SAT-solver is given in CNF. A CNF formula is a conjunction of *clauses*, where each clause is a disjunction of *literals*, and each literal is a single variable or its negation.

Propositional formulae over a ternary domain are encoded into Boolean logic by a dual-rail encoding [8], where each 3-valued variable is represented by two Boolean variables. With this encoding a 3-valued satisfiability problem can be solved by any Boolean SAT-solver.

III. Clock Gating

This section gives the necessary background on clock gating algorithms. It gives as an example one specific algorithm, and then presents the overall flow of how clock gating algorithms are used in practice.

Power consumption of digital circuits consists of *dynamic power*, which is the power consumed by circuit elements when they change state, and *static power* (or leakage), which is the power used by each circuit element when it is in a steady state. Clock gating decreases dynamic power by reducing the switching activity (the probability of changing value) of certain memory elements and parts of the clock network. This is achieved by preventing the clock input of memory elements from "ticking" when new values need not be loaded into them.

Clock gating techniques can be divided into two classes: combinational and sequential. Combinational clock gating algorithms disable the clock of latches whose outputs are not about to change ([24], [12], [23], [6], [13]). Sequential clock gating alters the behavior of latches without affecting design functionality, e.g. by identifying unused computations based on don't-care conditions ([9], [4], [1], [14]). In both cases, functional analysis is the strongest approach, since it guarantees to extract all clock gating opportunities for a given latch. From an algorithmic point of view, functional clock gating analysis is equivalent to constructing a combinational circuit (propositional function) C that represents potential clock gating opportunities of a certain type. Each input assignment that makes C evaluate to '1' is a clock gating opportunity. We show as an example one of the simpler algorithms and note that the results of this paper are equally applicable to all other functional analysis algorithms, e.g. [4], [24], [12], [23], [5], [14].

Consider the design in Figure 2(a). Obviously, the control signal of the MUX can be used as the clock gating condition

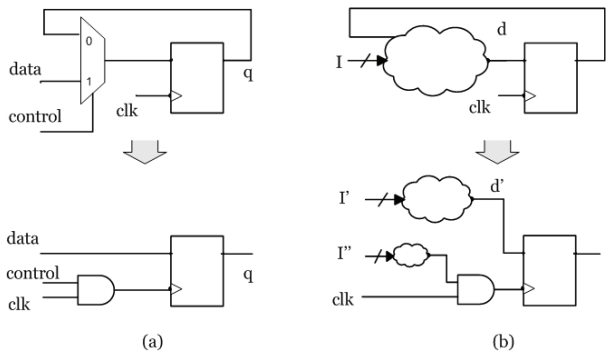


Fig. 2. Feedback loop elimination

for the given latch. In the more general case of Figure 2(b), we can build a BDD f_d for the signal d and then the clock gating function would be $f_{cg} = f_d \leftrightarrow q$, capturing exactly those states in which the value to be loaded into the latch (f_d) is identical to the one it currently holds (q), making a clock tick redundant. This produces a data-dependent clock gating function (a function that depends on q), which will likely violate timing constraints. To obtain a data independent clock gating function we need to universally quantify over q to get: ${f_{cg}}' = f_{cg}|_{q=0} \wedge f_{cg}|_{q=1}$.

This suggests the construction shown in Figure 3. In this setting the quantification of signal q is performed by creating two copies of the logic — one receiving q in a positive polarity and one receiving it in a negative polarity. For each copy we require the agreement between d and q, thus ensuring that (1) when the output o equals '1' the latch keeps its value and (2) this does not depend on the value of q. We term the construction of Figure 3 "Feedback Loop Elimination" (FLE).

Clock gating efficiency is measured by the probability of the clock gating function evaluating to '1', which is called the *on-set probability*. A function with high on-set probability is considered efficient since it gates the clock signal more often. In this sense, the construction of Figure 3 is the strongest data-independent clock gating function, since it describes a characteristic function for all the states (over I) in which the latch holds its value, regardless of the value of q.

In addition to requiring high on-set probability, there are

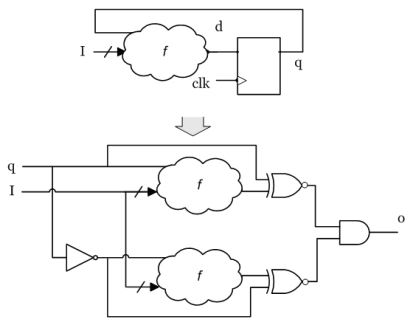


Fig. 3. The clock gating function constructed by the FLE algorithm

```
1: foreach latch l do
2:     C := Alg. construction for l
3:     cg(l) := BDD(C)
4: end for
5: find clock gating groups
6: foreach group g do
7:     b := BDD(cg(g))
8:     optimize(b, D)
9: end for
10: add clock gating functions to circuit
```

Fig. 4. General flow of clock gating synthesis

several constraints that clock gating functions must meet. In particular, they must not violate timing constraints or waste more power than they save (due to added leakage). Timing requirements usually stipulate that the logic depth of the generated circuitry should not exceed some constant. When the clock-gating function is represented in DNF, this constraint translates to a limit D on the number of literals in each term. To reduce the leakage overhead of the clock gating circuitry, latches with similar functions are grouped together and gated with a single function.

We examine clock gating synthesis algorithms that are all roughly based on the flow presented in Figure 4. For each latch in the design a combinational circuit C that represents potential clock gating opportunities of a certain type is constructed (line 2). The FLE construction presented above is just one option. Next, the BDD representation of C is built (line 3). This BDD is the strongest clock gating function (with respect to the chosen construction). After constructing the clock gating conditions for all latches, a grouping algorithm [3] is used to find sets of latches that can share a single clock gating function (line 5). This algorithm, which we do not discuss here, assigns a clock gating function `cg(g)` for each group of latches `g` that it creates. The aggregation may somewhat reduce the energy saving, as some latches may be gated with a function that is weaker than their computed gating function, but it significantly improves the overall power efficiency by reducing the amount of extra logic. Finally, the BDD of each gating function is built (line 7), and the function `optimize()` (line 8) optimizes it to generate a clock-gating function that adheres to the depth constraint D.

IV. Synthesis of clock gating functions

A. SAT-based approximation of gating functions

Our method is based on a combination of two techniques: representing clock gating opportunities as satisfying instances of a Boolean formula, and using a 3-valued abstraction to do universal quantification in order to minimize terms. For a given latch, a combinational circuit is built according to some clock gating algorithm, for example the FLE construction introduced in Section III. The propositional formula $f(I)$ of this circuit is the strongest clock gating function for this particular latch (and particular algorithm). The goal is to build a small approximation of this function, but without building its BDD and without calculating the unapproximated function.

Instead, we directly build a DNF formula representing the approximation. What we are looking for are bounded-size terms over I that imply $f(I) = 1$. Intuitively, our method uses a SAT-solver to enumerate over full assignments, but because we allow inputs to take on X-values we can extract from each assignment a smaller term by taking the conjunction of input values that are not X.

Let f be the propositional formula of a clock gating circuit C, and let I be the set of inputs to this circuit. For each $i \in I$ we add a fresh control variable α_i that determines whether i gets an X value or not. This is achieved by the following constraints:

$$\vartheta_f \triangleq \bigwedge_{i \in I} (\alpha_i \leftrightarrow (i \neq X))$$

Next, we add a cardinality constraint on the number of α_i's allowed to be 1:

$$\pi_f \triangleq (\sum_{i \in I} \alpha_i) \leq n$$

The constant n is chosen according to the size of terms we are looking for. The final formula built for f is:

$$\Psi_f \triangleq f \wedge \vartheta_f \wedge \pi_f$$

We hand Ψ_f to a SAT solver and get a (3-valued) satisfying assignment σ, from which we generate the corresponding term:

$$T(\sigma) \triangleq (\bigwedge_{i \in I.\sigma(i)=1} i) \wedge (\bigwedge_{i \in I.\sigma(i)=0} \neg i)$$

To proceed, we prevent the SAT solver from producing this assignment again by adding to its clause database the blocking clause $\neg T(\sigma)$. Note that this clause is relatively small (no more than n literals), and will block not only σ, but any satisfying assignment σ' that results in a term $T(\sigma')$ that is subsumed by $T(\sigma)$. The disjunction of terms $T(\sigma)$ for all satisfying assignments is an approximation of f with term-size less than or equal to n.

Figure 5 shows the pseudocode of `ApproxCGFunction`, our SAT-based algorithm for approximating the clock gating function of a single latch. The input is a propositional formula f and a bound n. The output is an approximation of f in DNF format, such that each term is of size n or less. The function $CNF()$ creates a CNF representation of its parameter. The CNF for f is created using the same algorithm that is typically used for bounded model checking [7], including many of the optimizations developed for this purpose. Ternary logic is implemented using a dual-rail encoding, in which each Boolean signal is represented by two variables. The CNF for the cardinality constraint is created using a SAT-efficient algorithm [20]. Obviously, we use incremental SAT-solving [22] for all calls in the loop of `ApproxCGFunction`. This greatly improves the performance of the solver and reduces the overhead of repeated calls. Finally, the halting condition (line 12) can be defined in many ways. We use a time limit in order to ensure that we do not spend too much time on a single latch (or group). The function res produced by `ApproxCGFunction` is an (under) approximation of the input f in that every satisfying assignment to res is also a satisfying assignment of f. If we set $n = |I|$ and remove the time limit, we have that $res \equiv f$.

ApproxCGFunction(f,n)

```
 1: F := CNF(Ψ_f)
 2: res := ∅
 3: done := false
 4: while not done do
 5:    σ := SAT(F)
 6:    if σ = ∅ then
 7:       done := true
 8:    else
 9:       F := F ∧ blocking_clause(σ)
10:       res := res ∨ T(σ)
11:    end if
12:    if comp_halt_cond() then
13:       done := true
14:    end if
15: end while
16: return res
```

Fig. 5. SAT-based approximation of a function

The loop in `ApproxCGFunction` resembles a naive all-SAT procedure, in that it repeatedly calls the SAT solver and then adds blocking clauses to prevent the solver from finding the same assignment twice. However, the convergence of this function is much faster because of the use of ternary logic. Each blocking clause that we add contains no more than n literals because each assignment contains no more than n explicit values. This significantly reduces the number of independent assignments to the formula, and hence the number of iterations required for reaching unsatisfiability.

Figure 6 shows the clock gating synthesis flow when using our SAT-based approximation method, which calls ApproxCGFunction where ever BDDs may explode. Given the circuit C that represents all clock gating opportunities for l, ApproxCGFunction is first called to produce an approximation f_l (line 3). After the grouping algorithm is invoked, the BDD for the clock gating function of each group (`cg(g)`) needs to be built. This step might also fail because the clock gating function of a group is the conjunction of the functions of all latches in the group. In the SAT-based flow we use ApproxCGFunction a second time, on the gating function of a group (line 7), before building the BDD and optimizing as before. Note that the BDD built on line 8 is for the approximated function, which is small and shallow and will not explode, thus avoiding the BDD size problem.

B. Over abstraction

In some cases, `ApproxCGFunction()` may not be able to find the minimal representation because of the abstraction we use. It is possible for there to be a term of size n that implies f while there exists no satisfying assignment to Ψ_f. This is because three-valued abstraction is an *approximation* of universal quantification, but is not exact. For example, consider the function $f \triangleq (i \vee \neg i) \wedge j$ and the assignment $\sigma(i, j) \triangleq (X, 1)$. Obviously, $j \rightarrow \forall i.f$, making j a suitable term for

```
1: foreach latch l do
2:     C := Alg. construction for l
3:     f_l := ApproxCGFunction(C,n)
4: end for
5: find clock gating groups
6: foreach group g do
7:     f_g := ApproxCGFunction(cg(g),n)
8:     b := BDD(f_g)
9:     optimize(b, D)
10: end for
11: add clock gating functions to circuit
```

Fig. 6. SAT-based flow of clock gating synthesis

clock gating, but $f(\sigma) \neq 1$. Since there is no satisfying assignment to Ψ_f with $n = 1$ the `ApproxCGFunction()` function cannot discover the minimal clock gating function on its own.

To manage this phenomenon we can use a slightly larger n than what is prescribed by the actual power efficiency limit D. In the above example, if we allow two non-X inputs we get the two satisfying assignments $\sigma_1(i,j) = (0,1)$ and $\sigma_2(i,j) = (1,1)$. Eventually we do build a BDD of the approximated function (Figure 6 line 8), and at this point σ_1 and σ_2 will merge into the minimal clock gating function j.

C. Minimizing term sizes

Our algorithm as presented so far does not necessarily find assignments with the minimal number of non-X values. We can force the SAT-solver to give us minimal terms (up to what is made possible by our use of abstraction) by adding an external loop that starts with $n = 1$ and increments n by one with each iteration, stopping at the desirable bound. In between iterations we do not restart the SAT-solver so that all blocking clauses from previous iterations remain. This ensures that there is no overlap between iterations, because if a term of size k is found the resulting blocking clause will suppress all larger terms that are subsumed by it. This method guarantees finding minimal terms, but incurs some overhead in the extra iterations, each ending with unsatisfiable instances. In general, unsatisfiable instances are harder to solve than satisfiable instances, so having to solve an unsatisfiable instance for each value of n that is tried out may prove to be expensive.

To avoid the penalty of unsatisfiable instances we may decide to use a time limit on the SAT-solver's run. If the time limit expires with no assignment we treat it as an "unsatisfiable" result and move on to the next iteration. Of course, this means that we are no longer guaranteed to generate minimal terms.

Another alternative for reducing the size of terms produced by the SAT-solver is to adjust the phase heuristic to favor injecting X-values when possible. The phase heuristic of a SAT-solver is the heuristic in charge of deciding whether to assign a decision variable with zero or one (given that the decision heuristic has already chosen a variable to decide upon). Whenever the phase heuristic is called it can choose the phase to favor the representation of X values. This heuristic will not guarantee finding minimal terms, but it significantly increases the likelihood of finding them while eliminating the need for an outer loop with several iterations.

V. Experimentation

We integrated our method into GateAlert, IBM's internal clock gating tool. The evaluation of its performance is based on applying `ApproxCGFunction` to the clock gating circuits produced by an FLE algorithm similar to that presented in Section III.

The first experiment we performed is aimed at quantifying the increase in gating that is achieved by the SAT-based method. To do this, we need to run our method on those latches for which the BDD analysis fails. Given the large amount of latches in modern designs, the processing time of a single latch must be limited, to prevent the entire analysis from becoming impractical. In our case, this time limit is 30 seconds per latch. Furthermore, based on experiments running BDD-based FLE on a large set of latches, sampled from numerous IBM designs, it was observed that 98% of the latches that can be solved by the BDD-based approach are solved within 15 seconds or less. Of the remaining 2% only 10% are solved within the 30 seconds time limit, which means that using a 15 seconds time limit halves the run-time while loosing only 0.2% of the results. We collected all latches from our data set for which BDD processing time exceeded 15 seconds. This set contains 5436 latches collected from 270 IBM designs. From here on we refer to these latches as *hard latches*.

Our benchmark data set contains blocks taken from IBM high-end processor designs, in which timing constraints are very tight, and the clause size limit is set to $n = 6$ (n being the cardinality constraint described in Section IV). In addition, the same memory limit as the BDD algorithm was given to the SAT approach (500MB), along with a time limit of 30 seconds per latch. The SAT-based approach succeeded in finding a clock gating condition for 3987 latches, which is more than 73% of all the hard latches [1]. For the remaining 27% hard latches, the SAT solver reported unsatisfiability. It should be noted that there were no instances for which the SAT solver reached the memory limit, and none in which it reached the time limit without giving any answer (either clock gating opportunity or unsatisfiability result).

To focus our analysis, we consider those designs in which there was a high rate of hard latches. Table I details only those designs for which the number of hard latches constituted more than 10% of the *total* number of candidate latches in the design (candidate meaning that the clock gating algorithm we are using is applicable to this latch). Note that the definition of a candidate depends on the clock gating algorithm so different algorithms will yield different numbers of candidates. In our experiments, only latches with a feedback loop are considered as candidates.

The next experiment is aimed at evaluating to what extent over-abstraction is hurting the results. As explained in Sec-

[1] Since the BDD method did not terminate on these latches we can not determine to what extent the result is an approximation of the optimum; more on this issue in the next experiment

TABLE I
PERFORMANCE OF THE SAT-BASED ALGORITHM ON THE DESIGNS WITH THE HIGH SKIPPED LATCHES RATIO.

Design	Candidates	Hard	% Hard	SAT solved	% Solved
D1	585	418	71.45%	418	100.00%
D2	576	273	47.40%	273	100.00%
D3	397	243	61.21%	243	100.00%
D4	1096	126	11.50%	126	100.00%
D5	234	72	30.77%	72	100.00%
D6	409	62	15.16%	62	100.00%
D7	328	60	18.29%	60	100.00%
D8	219	48	21.92%	48	100.00%
D9	1735	251	14.47%	250	99.60%
D10	626	134	21.41%	132	98.51%
D11	390	54	13.85%	53	98.15%
D12	212	99	46.70%	97	97.98%
D13	1580	202	12.78%	194	96.04%
D14	2507	270	10.77%	259	95.93%
D15	107	13	12.15%	10	76.92%
D16	247	54	21.86%	38	70.37%
D17	195	30	15.38%	5	16.67%

tion IV-B, it is possible that an unsatisfiable result is produced when in fact there are clock gating opportunities that could be described by a term of size n. We applied the BDD-based method to the set of latches for which the SAT solver gave an unsatisfiable result, call them un-solved latches, but this time on a stronger machine with much more time and memory, and with no limit on depth. For 2% of un-solved latches the BDD-based method proved that there are no gating opportunities, and for 78% of them BDDs still exploded, leaving us with 20% of unsolved latches in which gating opportunities are known to exist. However, the average depth of the clock gating function for these latches is 50.58. When approximating these clock gating functions down to depth 6 (corresponding to the term size limit used by the SAT-based procedure), only 2% of un-solved latches had clock gating opportunities, and their average on-set probability was as low as 0.09. Overall, 0.5% of hard latches were misses, in the sense that the SAT-based did not gate them although they could be gated if BDDs were allowed

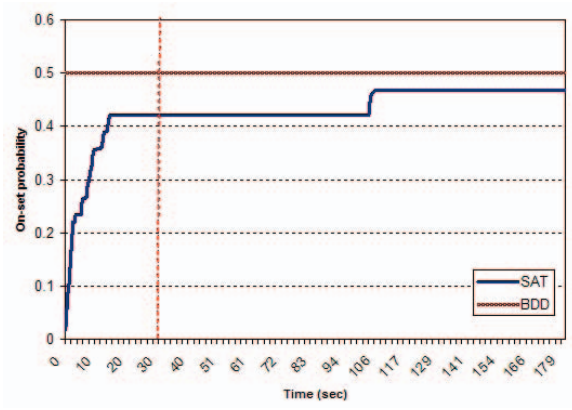


Fig. 7. The growth over time of the on-set probability of the approximation (for a single latch)

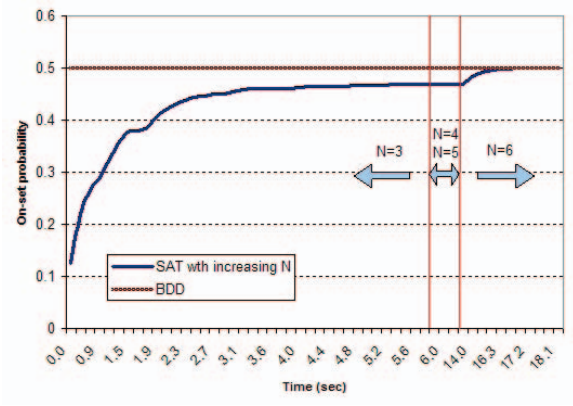


Fig. 8. The growth of the on-set probability of the approximation with increasing values of n

more memory and unlimited time. The low gating probability of these missed latches suggests that the odds of these gating functions making it to production are low; they would probably not be used due to power efficiency constraints.

The third experiment is meant to evaluate the impact of the SAT solver's timeout limit on the on-set probability of the resulting clock gating function. For this experiment we chose hard latches for which we were able to build the BDD (by increasing the memory resources available and removing the timeout limit) and computed the on-set probability for the clock gating function. The result for one representative latch is given in Figure 7. The optimum on-set probability in this case is 0.499514 (computed by the BDD). Our SAT-based approach achieves 85% (0.421509) of the optimum after 15 seconds and 94% (0.468292) of the optimum after 105 seconds. If allowed to continue, it takes more than 5 hours for the SAT solver to finish, arriving at a function with an on-set probability of 0.499512. The optimum on-set probability is achieved by a BDD of depth 33, while the SAT-based solution is of depth 6. BDD approximation methods that minimize the function to the required depth are heuristic by nature and may give different results. When approximating the BDD to depth 6 the resulting on-set probability is smaller, in some cases smaller than the result obtained by our procedure, which does not require any subsequent approximation. All the experiments (not shown here) on hard latches resulted in similar graphs, with varying degrees of smoothness to the function, and all justifying our choice of 30 seconds timeout limit for the SAT solver.

The final experiment evaluates whether it is worthwhile to start with $n = 1$ and increment by one each time we get UNSAT, compared to our previous results that only ran with $n = 6$. Figure 8 gives the results for the same latch used in Figure 7. Here, when $n = 1, 2$ the SAT solver returned an UNSAT result, thus the graph starts from $n = 3$. It took 6 seconds and 40 iterations for the SAT solver to reach an UNSAT result, and at this point the on-set probability of the computed approximation is 93% of the optimum (0.468292). Increasing n to 4 and 5 did not provide any satisfying assignments, and the UNSAT result is achieved very quickly.

When n reached 6, the SAT solver finished the computation within another 4 seconds (and 10 iterations), reaching an on-set probability of 0.499512. This function, of depth 6, is only 0.0002% below the optimum on-set probability that was achieved by a BDD of depth 33, and is computed in 18 seconds. Clearly, we cannot make sweeping claims based on a single example, but the results are consistent with the intuition that by forcing the SAT-solver to look for smaller terms we get stronger blocking clauses, covering the search space more quickly, and making the entire computation take less time. Overall, this small experiment suggests that increasing n is beneficial, although a more comprehensive study is required.

VI. Conclusion

We presented a SAT-based algorithm for synthesis of strongest clock gating functions. We circumvent the problem of BDD explosion when building the full clock gating function. This gives us the ability to handle much larger designs than before. Using 3-valued abstraction, we are able to directly generate the (strongest) approximation. Furthermore, our approach produces partial results even if the computation is not completed within a set time limit. We have shown that more than 70% of latches that could not be gated using the BDD-based approach can be gated by our SAT-based method, and that quality of the result is near optimum.

While motivated by a real EDA problem, the problem that we have defined and solved is a general problem of constrained satisfiability. Clearly, many insights from the use of of SAT solvers in bounded model checking may be transfered to this domain – the success of the iterative heuristic that gradually increments n is a good example of this. In future work we intend explore this and other optimizations, as well as the application to other clock gating algorithms.

References

[1] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-based sequential logic optimization for low power," in *ICCAD '94: the 1994 IEEE/ACM international conference on Computer-aided design*, 1994, pp. 74–81.

[2] F. A. Aloul, S. Hassoun, K. A. Sakallah, and D. Blaauw, "Robust sat-based search algorithm for leakage power reduction," in *PATMOS '02: 12th International Workshop on Power and Timing Modeling, Optimization and Simulation*, 2002.

[3] E. Arbel, C. Eisner, and O. Rokhlenko, "Resurrecting infeasible clock-gating functions," in *DAC '09: Proceedings of the 46th annual conference on Design automation*, 2009.

[4] P. Babighian, L. Benini, and E. Macii, "A scalable algorithm for RTL insertion of gated clocks based on ODCs computation," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, no. 1, pp. 29–42, 2005.

[5] L. Benini and G. D. Micheli, "Automatic synthesis of low-power gated-clock finite-state machines," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 630 – 643, 1996.

[6] L. Benini, G. D. Micheli, E. Macii, M. Poncino, and R. Scarsi, "Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers," *ACM Trans. Design Autom. Electr. Syst.*, vol. 4, no. 4, pp. 351–375, 1999.

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS'99: 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1999.

[8] R. E. Bryant and C. johan H. Seger, "Digital circuit verification using partially-ordered state models," in *In International Symposium on Multi-Valued Logic*, 1994.

[9] X. Chen, P. Pen, and C. L. Liu, "Desensitization for power reduction in sequential circuits," in *DAC '96: Proceedings of the 33rd annual conference on Design automation*. New York, NY, USA: ACM, 1996, pp. 795–800.

[10] M. Fahim Ali, A. Veneris, A. Smith, S. Safarpour, R. Drechsler, and M. Abadir, "Debugging sequential circuits using boolean satisfiability," in *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, 2004, pp. 204–209.

[11] O. Grumberg, A. Schuster, and A. Yadgar, "3-valued circuit sat for ste with automatic refinement," in *ATVA '07: 5th International Symposium on Automated Technology for Verification and Analysis, Tokyo, Japan*, 2007, pp. 457–473.

[12] A. P. Hurst, "Automatic synthesis of clock gating logic with controlled netlist perturbation," in *DAC '08: Proceedings of the 45th annual conference on Design automation*, 2008.

[13] H. M. Jacobson, "Improved clock-gating through transparent pipelining," in *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2004, pp. 26–31.

[14] H. Kapadia, L. Benini, and G. D. Micheli, "Reducing switching activity on datapath buses with control-signal gating," *IEEE Journal of Solid-State Circuits*, vol. 34, pp. 405–414, 1999.

[15] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large boolean functions via interpolation and satisfiability solving," in *DAC '08: Proceedings of the 45th Design Automation Conference, Anaheim, CA, USA*, 2008.

[16] A. Mishchenko and R. K. Brayton, "Sat-based complete don't-care computation for network optimization," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 412–417.

[17] J.-W. Roorda and K. Claessen, "A new sat-based algorithm for symbolic trajectory evaluation," in *CHARME '05: 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Saarbrücken, Germany*, 2005.

[18] J. W. Roorda and K. Claessen, "Sat-based assistance in abstraction refinement for symbolic trajectory evaluation," in *CAV '06: 18th International Conference on Computer Aided Verification*, 2006.

[19] T. Schuele and K. Schneider, "Three-valued logic in bounded model checking," in *MEMOCODE '05: the 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, 2005, pp. 177–186.

[20] C. Sinz, "Towards an optimal cnf encoding of boolean cardinality constraints," in *CP'05: 11th International Conference on Principles and Practice of Constraint Programming*, 2005.

[21] S. Staber, G. Fey, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," in *HVC'06: Haifa Verification Conference*, 2006, pp. 50–64.

[22] O. Strichman, "Pruning techniques for the sat-based bounded model checking problem," in *CHARME'01: 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2001.

[23] F. Theeuwen and E. Seelen, "Power reduction through clock gating by symbolic manipulation." in *Proc. IFIP Int. Workshop on Logic and Architecture Synthesis*, 1996, pp. 184–191.

[24] R. Wiener, G. Kamhi, and M. Y. Vardi, "Intelligate: Scalable dynamic invariant learning for power reduction," in *PATMOS '08: 18th International Workshop on Power and Timing Modeling, Optimization and Simulation*, ser. LNCS, vol. 5349, 2008.

[25] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "Sat sweeping with local observability don't-cares," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*. New York, NY, USA: ACM, 2006, pp. 229–234.

Finding heap-bounds for hardware synthesis

B. Cook, MSR
A. Gupta, MPI-SWS
S. Magill, CMU
A. Rybalchenko, MPI-SWS
J. Simsa, CMU
S. Singh, MSR
V. Vafeiadis, MSR

***Abstract*—Dynamically allocated and manipulated data structures cannot be translated into hardware unless there is an upper bound on the amount of memory the program uses during all executions. This bound can depend on the *generic parameters* to the program, *i.e.*, program inputs that are instantiated at synthesis time. We propose a constraint based method for the discovery of memory usage bounds, which leads to the first-known C-to-gates hardware synthesis supporting programs with non-trivial use of dynamically allocated memory, *e.g.*, linked lists maintained with `malloc` and `free`. We illustrate the practicality of our tool on a range of examples.**

I. Introduction

C-to-gates synthesis promises to bring the power of hardware based acceleration to mainstream programmers and to radically increase the productivity of digital designers [17]. However, today's C-to-gates synthesis tools do not support one of the most powerful and widely used features of high-level programming in C—dynamically allocated data structures. This leads to the use of arrays and significantly more complicated code for modelling naturally dynamic data structures with static data structures, which in turns incurs extra cost due to the extra complexity of design, verification, and maintenance. The support for dynamic memory abstraction remains an on-going research problem because of the need to efficiently and accurately determine a bound on heap consumption.

This paper advances the state-of-the-art in hardware synthesis by providing support for programs that dynamically allocate, deallocate, and manipulate heap-based data structures. Our technical contribution is a constraint-based method for finding a symbolic bound on the maximum heap size at compile time. This symbolic bound is expressed as a linear function on the *generic parameters* to the circuit description.[1] With our method for computing symbolic bounds we can then automatically translate C programs with dynamic memory usage into equivalent programs that operate over statically allocated arrays. That is, when circuit descriptions are instantiated in their surrounding designs, the symbolic bounds can be used to compute concrete bounds for use during synthesis.

Our method significantly increases the expressive power available to the users of synthesis systems. For example, with our new C-to-gates synthesis flow, a designer can think in terms of a tree-based data structure, yet generate hardware that operates on a flat fixed sized array. Furthermore, off-the-shelf libraries can now be used as subroutines by digital designers. This leads to better re-use, as well as new avenues of adapting software verification techniques for use in hardware systems.

Our experiments show that it is possible to produce viable circuits from C programs that use dynamic data structures. By viable we mean that the synthesized circuits have performance that is good enough so that we see a possibility to significantly improve it with future work. This claim needs empirical justification by producing and analyzing the hand-coded equivalents. However, the generated circuits have a size and operating frequency which seems quite plausible in the opinion of one of the authors, who has experience of producing highly optimized FPGA designs.

Related work. C-to-gates synthesis is a maturing field with notable systems—see [6], [7], [13], [18], [21], [26], [33], [34]. Some existing C-to-gates synthesis systems already support pointers and pointer aliasing, see *e.g.* [32], but they do not deal with dynamically allocated data structures.

Synthesis tools for other general purpose programming languages also exist (*e.g.* tools supporting Scheme [30], or Haskell [3]). In a few rare instances (*e.g.* [5]) tools have been used not only to generate hardware but also the circuit's correctness proof as well. These tools usually require the user to estimate the maximal amount of memory allocated by the program and take this quantity as an input parameter to the synthesis routine. Thus, the results of our work can perhaps be used with these existing tools.

In the domain of pure functional programming languages, the topic of heap-bounds analysis has been extensively investigated, see *e.g.* [19]. For imperative programs, [20] develops a type system which tracks memory consumption. The Java memory-bounds tool described in [1] uses a heap abstraction and applies heuristics based on arithmetic simplification to find a memory bound. In contrast, our method uses a more precise numerical abstraction for dealing with heap, as we keep track of the size of intermediate list segments identified by the shape analysis when dissecting the heap, which was crucial for dealing with our examples. Furthermore, instead of using heuristics for finding the bound expression, we apply a constraint based boundedness analysis which is complete for linear bound expressions provable using linear invariants.

The semi-manual technique proposed in [4] uses the Daikon [11] to collect likely program invariants—including facts about memory consumption—and uses them to derive an initial set of bound candidates.

In principle, the existing techniques for proving computational complexity, *e.g.* [14], can be used as a basis to design an algorithm for discovery of memory usage bounds. However,

[1]The term generic parameter is used in hardware design languages to describe variables whose values will be known at compile-time.

978-1-4244-4966-8/09/$25.00 ©2009 IEEE

```
void prio(int n,in_signal i,out_signal o) {
  LINK *tmp,*c,*buffer;
  assert( n>0 );
  while (1) {
    buffer = NULL;
    // Build up an n-sized sorted buffer
    for (int k=0;k<n;k++) {
      buffer = sorted_insert(input(i),buffer);
    }
    // Send the sorted list to the output and
    // deallocate the buffer as we walk it
    c=buffer;
    while(c!=NULL) {
      output(o,c->data);
      tmp = c;
      c = c->next;
      free(tmp);
    }
  }
}
```

Fig. 1. Priority queue circuit specification in C, using off-the-shelf implementation of `sorted_insert`. The *generic* parameter `n` is assumed to be specified at compile-time.

```
LINK * sorted_insert(int data, LINK *l) {
  LINK * elem = l;
  LINE * prev = NULL;
  LINK * x = (LINK*)malloc(sizeof(LINK));
  assert(x!=NULL);
  x->data = data;
  while (elem != NULL) {
    if (elem->data >= x->data) {
      x->next = elem;
      if (prev == NULL) { l = x; return l; }
      prev->next = x;
      return l;
    }
    prev = elem;
    elem = elem->next;
  }
  x->next = elem;
  if (prev == NULL) { l = x; return l; }
  prev->next = x;
  return l;
}
```

Fig. 2. Off-the-shelf implementation of incremental insertion sort procedure.

since we are only interested in bounds expressed over generic parameters, a major challenge is to bias the bound discovery method towards such well-formed bounds. Our constraint based procedure solves this challenge.

Our approach for finding symbolic bounds uses several known methods and tools as sub-procedures, such as shape analysis (*e.g.* [10], [23], [25]) and abstraction methods based on the introduction of new variables (*e.g.* [22], [24]). Our new constraint-based method draws influence from previously developed methods for invariant generation and rank function synthesis (*e.g.* [9], [31]).

II. Example

Imagine that we would like to build an `n`-size priority queue circuit that reads integers from an input signal and returns every `n` input integers on an output signal in sorted order. See the function `prio` in Fig. 1 for an example of how we might wish to write a specification of the desired hardware in C. Our intention is that the variable `n` in Fig. 1 is a generic parameter, whereas `i` and `o` should be thought of as signal names. Our synthesis tool treats these in a special way as standard C, of course, does not make this distinction. In this example we assume that the circuit uses `input()` and `output()` as primitives for I/O on the signal variables `i` and `o`. `LINK` is a C struct used to represent singly-linked lists (with fields `data` and `next`). We make use of an existing off-the-shelf insertion-sort implementation, `sorted_insert`. See Fig. 2 for the source code of `sorted_insert`.

Note that in order to convert this program into hardware we must first find an *a priori* bound on the amount of heap during the execution of `prio`, for any input or parameter. The problem is that `sorted_insert` does not guarantee a concrete bound on the amount of heap allocated during its execution, instead it preserves a bound – it takes a state where k heap cells have been allocated and returns a state in which $k+1$ have been allocated. Thus we must hope to find a bound on the amount of heap used by `sorted_insert` from states limited to those reachable from `prio`.

If we can find this bound, then we can convert the program's operations on the heap into operations on statically-allocated arrays, thus facilitating synthesis. We aim to find a bound that holds across the entire program, but is expressed symbolically using only the generic parameters to the top-level function (*i.e.* the parameter `n` of the circuit `prio`). This allows us to pre-allocate a shared array when creating instances of the circuit `prio`.

The procedure given later in Section III is designed to find a function f such that it is a program invariant that $f(\mathtt{n})$ is larger than the number of heap cells allocated at any given time during its execution. In this case the procedure described later will find the function $f(\mathtt{n}) = \mathtt{n} * 8$, assuming that `sizeof(LINK)` $= 8$ in the encoding.

With f we can now re-encode the program using a pre-allocated array. In essence, when we know the valuations to the input parameters we can then pre-allocate an array using f. We then convert dereferences like `*c` into `a[c]`. Field offsets are explicitly encoded: `c->data` is encoded as `a[c+0]`, and `c->next` is encoded as `a[c+4]`.

From this program (and via a translation into VHDL) we then used the Altera Quartus II 9.0 tools to construct an implementation for the Stratix III FPGA architecture. Using default synthesis and implementation options and with `n` $= 10$, the generated circuit uses 5859 adaptive look-up tables, 4598 logic registers and 8192 block memory.

III. From heaps to arrays

In this section we describe an analysis that automatically discovers symbolic bounds on the heap usage. We will assume

that the size parameters passed to `malloc` are fixed constants. Through the use of static analysis, we annotate each call to `free` with the amount of memory the call is freeing. For example, we would transform the call `free(tmp)` from Fig. 1 to `free(tmp,sizeof(LINK))`. For simplicity of presentation we will assume that programs allocate and free heap cells of a single fixed size. We can support multiple size allocations through the use of compile-time partial evaluation, but at the cost of complexity in the notation in this section. We currently do not support arbitrary DAGs or hash-tables, due to the limitations of existing separation logic based shape analysis tools [8], [10], [23], [25] of which we are dependent.

Our procedure is divided into the following steps.

a) Numerical heap abstraction: First, we augment the program with a new variable h, which is used to track the amount of heap that is currently allocated. The variable h is incremented when malloc is called, and decremented when free is called. For memory-safe programs such behavior of h is correct. We use the shape analysis tool THOR [25] to determine the shape of the data structures used during the program's execution, and to prove memory safety. Using techniques from [24], THOR can be used to produce a new program without heap that is a sound abstraction of the original program—additional integer variables are added by THOR to summarize the sizes of data-structures. Thus, bounds found on h in the abstraction imply bounds in the original program. Note that the new program variables range over integers of arbitrary size (*i.e.* they cannot be represented in 32 or 64 bits).

The new abstract program is used for computing bounds on heap consumption only, and does not play any role during the hardware synthesis step.

b) Numerical bounds analysis: Next, we apply our constraint-based boundedness analysis to the numeric program to find a symbolic bound f on the maximum value of h. For improved scalability we combine our constraint-based synthesis approach with a counterexample-guided method of checking and refining candidate bounds.

c) Array-based heap management and synthesis: Once we have computed a symbolic bound (assuming that a bound can be found) we throw away the abstraction and then convert the original program into an array-based program operating over a pre-allocated shared array and then apply off-the-shelf synthesis tools to produce a gate-level design. Note that, although we may sometimes compute a conservative over-approximation for a bound on memory usage, it is often the case that a downstream synthesis tool can perform further pruning to yield a gate level implementation that does indeed have a better (or even ideal) bound. A simple case of this scenario is when a list is used to represent a bit-vector which is used in arithmetic expressions with known range at synthesis time allowing some of the upper bits to be pruned.

The following sections discuss the above procedures in more detail.

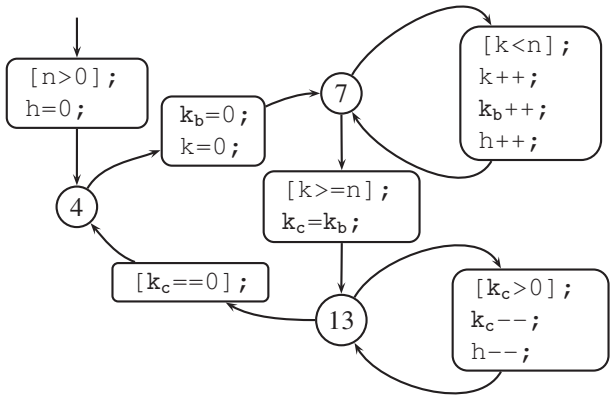


Fig. 3. Numerical abstraction of procedure `prio` shown from Fig. 1. Commands of the form `[e];` denote assume statements.

IV. NUMERICAL HEAP ABSTRACTION

A shape analysis tool is designed to take a program and compute an invariant for each program location describing the shape of the heap. The invariant describes the data structures stored in the heap during the program's execution. Shape analysis tools are based on symbolic simulation together with abstraction techniques.

Using techniques described in [24], the shape analysis tool THOR can be used to introduce new variables which soundly track the sizes of data structure shapes inferred by the shape analysis. In the example of the function `prio`, THOR would introduce a variable $\mathtt{k_b}$ recording the length of the linked list starting from `buffer`. At the command `buffer = NULL`, we initialize $\mathtt{k_b}$ to zero. At the lines `prev->next = x` within `sorted_insert`, the length of that linked list is increased; therefore the abstraction will increment $\mathtt{k_b}$. Similarly, THOR will introduce another variable $\mathtt{k_c}$ recording the length of the linked list from `c`. Corresponding to the assignment `c=buffer`, the abstraction will set $\mathtt{k_c}$=$\mathtt{k_b}$, and at the assignment `c=c->next`, the abstraction decrements $\mathtt{k_c}$. Also, when we exit the `while(c!=NULL)` loop, we know that `c==0`, and hence also $\mathtt{k_c}$=0.

Fig. 3 shows the control-flow graph (CFG) of the resulting abstraction of `prio`. The CFG contains three nodes corresponding to the three loops in the `prio` function. These nodes are connected by the edges which are annotated with the code occurring between the locations. The transitions between locations come in two forms: assignments v=e`;` and assumption checks `[`e`];`. The assumptions prune executions in which the condition e does not hold.

For brevity, calls to the function `sorted_insert` in Fig. 3 have been summarized as the transition $\{\mathtt{k_b}\mathtt{++;h++;}\}$ from location 7 to 7, but our technique is designed to work for a fully expanded CFG of the code.

V. NUMERICAL BOUNDS ANALYSIS

Preliminaries. Our shape analysis procedure produces a program $\mathcal{P} = (V, h, P, \mathcal{L}, \ell_{init}, \mathcal{T})$ that consists of a set of variables V, a heap consumption variable $h \in P$, a set of

generic parameters $P \subseteq V \setminus \{h\}$, a set of locations $\mathcal{L}$, an initial location $\ell_{init} \in \mathcal{L}$ and a set of abstract transitions $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is given by a tuple (ℓ, ρ, ℓ') where $\ell, \ell' \in \mathcal{L}$ and ρ is a constraint over $V \cup V'$, where the variables in V' represent the values of variables V after the transition is executed. Each transition relation preserves the values of generic parameters, *i.e.*, for each $(\ell, \rho, \ell') \in \mathcal{T}$ we have

$$\forall V \; \forall V' : \rho \rightarrow P' = P \; .$$

A *state* s is a valuation of V. A *computation* is a sequence of location and state pairs $(\ell_1, s_1), (\ell_2, s_2), \ldots$ such that ℓ_{init} is the initial location, *i.e.*, $\ell_1 = \ell_{init}$, and for each consecutive pair (ℓ_i, s_i) and (ℓ_{i+1}, s_{i+1}) there is a transition $(\ell_i, \rho, \ell_{i+1}) \in \mathcal{T}$ such that $(s_i, s_{i+1}) \models \rho$. A state s is reachable at a location ℓ if the pair (ℓ, s) appears in some computation.

An *invariant* at a location $\ell \in \mathcal{L}$ is a superset of all reachable states at ℓ. We represent invariants by formulas over the variables V. An *invariant map* Inv assigns an invariant to each location. In particular, we have $Inv(\ell_{init}) = true$, *i.e.*, every state is reachable at the initial location. We will use primed notation $Inv(\ell)'$ for $Inv(\ell)[V'/V]$. An invariant map Inv is *parametric* if it does not restrict the values of program variables besides the generic parameters and the heap consumption variable, *i.e.*, for each $\ell \in \mathcal{L}$ we have

$$\forall V : Inv(\ell) \leftrightarrow (\exists V \setminus (P \cup \{h\}) : Inv(\ell)) \; .$$

An invariant map Inv is *inductive* if for each program transition $(\ell, \rho, \ell') \in \mathcal{T}$ we have

$$\forall V \; \forall V' : \; (Inv(\ell) \wedge \rho) \rightarrow Inv(\ell')' \; .$$

We are interested in a parametric invariant map Bnd that bounds the heap consumption. Formally, we will search for Bnd such that for each $\ell \in \mathcal{L}$ we have

$$\forall P \; \exists c \in \mathbb{N} \; \forall V \setminus P : Bnd(\ell) \rightarrow h \leq c \; .$$

Then, the maximal value of the constant c among all program locations determines the maximal amount of memory that is dynamically allocated during the program computation.

For proving that Bnd is valid we will need an inductive invariant map Inv. Formally, we require that for each $\ell \in \mathcal{L}$ holds

$$\forall V : Inv(\ell) \rightarrow Bnd(\ell) \; .$$

Bounds analysis algorithm. Fig. 4 presents our constraint-based procedure BOUND for discovering heap consumption bounds. The procedure takes as parameters a program $\mathcal{P}$, an invariant template map Inv^T, and a bound template map Bnd^T. It returns either a valid bound map or an exception if no such map can be found.

The template maps used by BOUND are reminiscent of those used in constraint-based invariant generation [9], [31] and rank function synthesis [27]. A template map assigns an assertion over program variables and *template* parameters to each program location. The template map Inv^T may use a template of the form

$$\alpha_1 v_1 + ... + \alpha_n v_n \leq \alpha \wedge \beta_1 v_1 + ... + \beta_n v_n \leq \beta \; ,$$

which is a conjunction of two linear inequalities with the template parameters $\alpha_1, ..., \alpha_n, \alpha, \beta_1, ..., \beta_n, \beta$ and program variables $V = \{v_1, \ldots, v_n\}$.

The bound template map Bnd^T given to BOUND as input assigns to each program location a bound template of the form

$$h \leq \delta_1 p_1 + \cdots + \delta_m p_m + \delta \; ,$$

where $\delta_1, \ldots, \delta_m, \delta$ are *template* parameters and $P = \{p_1, \ldots, p_m\}$ are *generic* parameters. Since Bnd^T only refers to P and h, it guarantees to yield parametric bound invariants only.

BOUND collects a conjunction of constraints Ψ over template parameters for both template maps in lines 1–5. These constraints encode the condition that the computed bounds must be valid. Lines 2–3 state that the bounds hold for all reachable states, which are represented by an invariant map induced by the invariant template map Inv^T. Lines 4–5 encode the condition that Inv^T in fact represents all reachable program states.

We collect all template parameters in line 6. If our constraint solving procedure can find a satisfying assignment to Ψ, then this assignment defines a bound map in line 8. Otherwise, BOUND raises an exception.

The transition relations in the program $\mathcal{P}$ produced during the shape analysis phase are conjunctions of linear inequalities over V and V'. For our templates consisting of linear inequalities, we eliminate the universally quantification over V and V' in lines 3 and 5 of BOUND by applying a standard technique,

procedure BOUND
input
  $\mathcal{P} = (V, h, P, \mathcal{L}, \ell_{init}, \mathcal{T})$: program
  Inv^T: invariant template map
  Bnd^T: bound template map
var
  Q: template parameters in Inv^T and Bnd^T
  Ψ: auxiliary constraint over Q
begin
1 Ψ := $true$
2 **foreach** $\ell \in \mathcal{L}$ **do**
3 $\Psi := \Psi \wedge \forall V : Inv^T(\ell) \rightarrow Bnd^T(\ell)$
4 **foreach** $(\ell, \rho, \ell') \in \mathcal{T}$ **do**
5 $\Psi := \Psi \wedge \forall V \; \forall V' : (Inv^T(\ell) \wedge \rho) \rightarrow Inv^T(\ell')'$
6 Q := free variables in Ψ
7 **if** exists M such that $\Psi(M)$ **then**
8 **return** $Bnd^T[M/Q]$
9 **else**
10 **raise** "no bound found"
end

Fig. 4. BOUND discovers bounds on the value of the variable h, which keeps track of the amount of dynamically allocated memory.

see *e.g.* [9], based on Farkas' lemma [12]. The resulting constraint Ψ is a conjunction of non-linear inequalities and can be efficiently solved using the existing tools, *e.g.* [15], [16]. We implemented our algorithm in the ARMC model checker [28].

The soundness and completeness of BOUND is formalized in the following theorem.

Theorem 1. *The procedure* BOUND *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e.*, in this case it computes a bound map. The procedure* BOUND *is also sound,* i.e.*, it computes a bound map that represents an upper bound on the memory usage.*

Proof: (Sketch) We rely on the soundenes and completeness of the translation of the bounds synthesis problem to constraint solving. The translation follows the classical scheme applied for the synthesis of inductive invariants using constraint solving. ■

Example. Consider the program in Fig. 3 over the variables $\mathtt{n}$, h, $\mathtt{k}$, $\mathtt{k_b}$, and $\mathtt{k_c}$. The only generic parameter is the variable $\mathtt{n}$.

We consider a template map Inv^T that assigns to each program location a conjunction of two linear inequalities. For example, for the location ℓ_7 we have

$$\begin{aligned} Inv^T(\ell_7): \quad & \alpha_{\mathtt{n}}\mathtt{n} + \alpha_h h + \alpha_{\mathtt{k}}\mathtt{k} + \alpha_{\mathtt{k_b}}\mathtt{k_b} + \alpha_{\mathtt{k_c}}\mathtt{k_c} \leq \alpha \wedge \\ & \beta_{\mathtt{n}}\mathtt{n} + \beta_h h + \beta_{\mathtt{k}}\mathtt{k} + \beta_{\mathtt{k_b}}\mathtt{k_b} + \beta_{\mathtt{k_c}}\mathtt{k_c} \leq \beta \wedge \\ & \gamma_{\mathtt{n}}\mathtt{n} + \gamma_h h + \gamma_{\mathtt{k}}\mathtt{k} + \gamma_{\mathtt{k_b}}\mathtt{k_b} + \gamma_{\mathtt{k_c}}\mathtt{k_c} \leq \gamma \end{aligned}$$

The bound template at this location is

$$Bnd^T(\ell_7): \quad h \leq \delta_{\mathtt{n}}\mathtt{n} + \delta \ .$$

Next, BOUND creates a conjunction of constraints Ψ over the template parameters from all program locations. We only present two constraints from Ψ that are created at lines 3 and 5 for the location ℓ_7 and the loop transition at the location ℓ_7 respectively. The first constraint is the implication

$$\forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c} : Inv^T(\ell_7) \rightarrow Bnd^T(\ell_7) \ .$$

The second constraint involves the transition relation of the loop:

$$\begin{aligned} & \forall \mathtt{n}\ \forall h\ \forall \mathtt{k}\ \forall \mathtt{k_b}\ \forall \mathtt{k_c}\ \forall \mathtt{n}'\ \forall h'\ \forall \mathtt{k}'\ \forall \mathtt{k}'_{\mathtt{b}}\ \forall \mathtt{k}'_{\mathtt{c}} : \\ & \quad (Inv^T(\ell_7) \wedge \\ & \quad \mathtt{k} < \mathtt{n} \wedge \mathtt{n}' = \mathtt{n} \wedge h' = h + 1 \wedge \mathtt{k}' = \mathtt{k} + 1 \wedge \\ & \quad \mathtt{k}'_{\mathtt{b}} = \mathtt{k_b} + 1 \wedge \mathtt{k}'_{\mathtt{c}} = \mathtt{k_c}) \rightarrow \\ & \quad Inv^T(\ell_7)' \end{aligned}$$

We solve Ψ and obtain $\delta_{\mathtt{n}} = 1$ and $\delta = 0$ for the bound template parameters occurring in the location ℓ_7, *i.e.*, we have

$$Bnd^T(\ell_7) = (h \leq \mathtt{n}) \ .$$

The corresponding invariant map assigns $h \leq \mathtt{k_b} \wedge \mathtt{k_b} \leq \mathtt{k} \wedge h \leq \mathtt{n}$ to the location ℓ_7. In our example, the bound occurs in the corresponding inductive invariant; in general, however, this need not be the case.

```
   procedure INCBOUND
   input
     P = (V, h, P, L, ℓ_init, T): program
     Inv^T: invariant template map
     Bnd^T: bound template map
   var
     Bnd : bound map
     ℓ_err: distinguished error location
     T_E: transitions for bound assertion checking
   function PATHPROGRAM
   input
     π : sequence of transitions
   begin
1    return (V, h, P, L, ℓ_init,
2            {τ | τ = (ℓ, ρ, ℓ') occurs in π and ℓ' ≠ ℓ_err})
   end;
   begin
3    Bnd := λℓ ∈ L.h ≤ 0
4    repeat
5      T_E := {(ℓ, ¬Bnd(ℓ) ∧ V' = V, ℓ_err) | ℓ ∈ L}
6      if exists π ∈ (T ∪ T_E)* from ℓ_init to ℓ_err
7        such that ρ_π ≠ ∅ then
8        P_π := PATHPROGRAM(π)
9        try
10         Bnd_π := BOUND(P_π, Inv^T, Bnd^T)
11       catch
12         return "unbounded consumption path π"
13       Bnd := λℓ ∈ L.Bnd(ℓ) ∨ Bnd_π(ℓ)
14     else
15       return "bound assertion map Bnd"
16   done
   end
```

Fig. 5. INCBOUND performs an incremental boundedness analysis using guidance from spurious counterexamples.

Incremental bounds analysis. The constraint-based procedure BOUND performs an expensive computation—non-linear constraint solving—and does not scale beyond medium-sized programs. We improve the scalability of BOUND by performing the boundedness analysis in an incremental fashion using the idea of path invariants [2]. We apply the expensive, constraint-based procedure only to certain program fragments, which are determined automatically.

Fig. 5 presents our BOUND-based procedure INCBOUND for an incremental discovery of heap consumption bounds for the full program from its fragments. Initially, the bound map states that no heap consumption takes place, see line 3. Then, this claim is verified in lines 6–7 using a verification tool for proving program safety. Such a tool is applied on an augmented program that is obtained from $\mathcal{P}$ by adding a distinguished error location ℓ_{err} that is reachable if the heap bound claimed by Bnd is not valid. In the case of a false bound, the algorithm will return a counterexample in the form of a sequence of transitions π that leads to heap consumption

beyond the claimed bound.

In case a counterexample π is found, we identify a fragment of $\mathcal{P}$ that is traversed by the transitions occurring in π. This code fragment is defined by a path program $\mathcal{P}_\pi$ for π [2], see lines 1–2. In particular, the path program $\mathcal{P}_\pi$ traverses the same loops of $\mathcal{P}$ that are visited by π.

We compute an adjustment Bnd_π for the bound map by applying the procedure BOUND on the path program, see line 10. The adjustment is used to weaken the claimed bound, see line 13.

This sequence of incremental adjustments continues until either the full program $\mathcal{P}$ satisfies the claimed bound map or a path that for which no heap consumption bound can be found is discovered.

The soundness and completeness properties of INCBOUND are inherited from the procedure BOUND and the notion of path invariants.

Theorem 2. *The procedure* INCBOUND *is complete for bound expressions in linear arithmetic provable using linear arithmetic invariants,* i.e.*, in this case it computes a bound map and terminates. The procedure* INCBOUND *is also sound,* i.e.*, it computes a bound map that represents an upper bound on the memory usage.*

Proof: (Sketch) We rely on the fact that the computed path programs grow by at least one transition at each iteration. Once all program transitions appear in the path program, Theorem 1 applies. ■

Example. Consider finding a bound for h in the program from Fig. 3. In the algorithm from Fig. 5 we start with a candidate bound $h \leq 0$ at each location. We can then attempt to prove that $h \leq 0$ at every location using a symbolic model checker (this corresponds to lines 5-7 of Fig. 5. In this case $\mathtt{h} \leq 0$ is not necessarily true at location 7 in Fig. 3, in which case the symbolic model checker will return a witness counterexample path. Imagine that we get the path $\pi = 4 \rightarrow 7$. In this case PATHPROGRAM(π) will return a sub-program of Fig. 3, as found in Fig. 6. We can then find a bound on this sub-program, resulting in $\mathtt{h} \leq \mathtt{n}$. Thus, we refine the candidate whole-program bound to be $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$. Repeating the steps from lines 5-7 allows us to prove that $\mathtt{h} \leq 0 \vee \mathtt{h} \leq \mathtt{n}$ is a valid bound for the whole program. After simplification, we return $\mathtt{h} \leq \mathtt{n}$.

VI. ARRAY-BASED HEAP MANAGEMENT

Numerical boundedness analysis computes a bound on the maximal amount of memory that is dynamically allocated during program computation, and represents this bound as a function of generic parameters. When synthesizing a hardware implementation, the generic parameters are instantiated. Hence we obtain a concrete bound, say N.

Next, we replace all heap operations in the program $\mathcal{P}$ by operations on a statically allocated array `a` of size N. Each pointer to the heap becomes an array index. Field accesses are converted into arithmetic operations over array indices. For

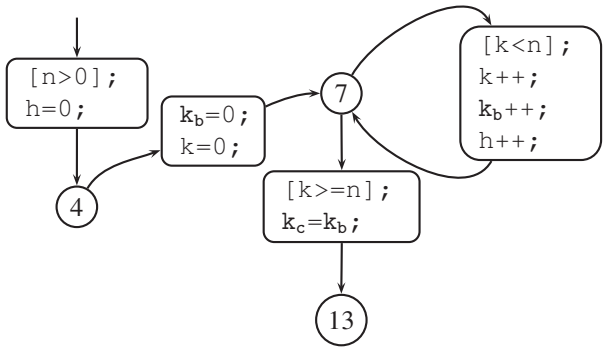


Fig. 6. Path program for the program from Fig. 3 and a path consisting of transitions between the locations (ℓ_{init}, ℓ_4), (ℓ_4, ℓ_7), (ℓ_7, ℓ_7), and (ℓ_7, ℓ_{13}).

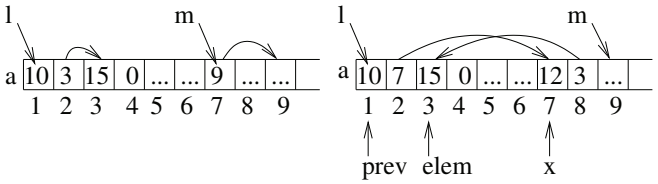


Fig. 7. Creation of a new `LINK` structure in the array-based heap implementation.

example, the statement `c = c->next;` from the program in Fig. 1 becomes `c = a[c+4];`, where the offset 4 is due to the four byte size of an array cell.

We use a list of array indices that is embedded into the array `a` to keep track of free array cells. Each list element is an index of a free cell. We introduce a global variable `m` that stores the head of the list, and hence the cell at index `m` is free. Then, the value of `a[m]` is the next list element, which is the index of the second free cell stored in the list. We obtain the third element by accessing `a[a[m]]` and so on. Initially $\mathtt{m} = 0$ and the array `a` is initialized in the following way:

$$\forall 0 \leq i < N : \mathtt{a}[i] = i + 1 \ .$$

A call to `malloc()` consumes the head of the list. That is, `x = malloc()` is implemented by the sequence of instructions `x = m; m = a[m];`, where the first assignment delivers the free cell and the second assignment ensures that the subsequent call to `malloc` will return the next free cell in the list. We do not need to check whether the free list empty because the boundedness analysis guarantees that it will never happen, i.e., we have $m \leq N$.

Fig. 7 illustrates the array-based treatment of `malloc`. We assume that the heap stores data structure `LINK`, whose size is two integers, and that each array cell is of size one integer. The array on the left is free starting at the index 7, as represented by the valuation $\mathtt{m} = 7$, $\mathtt{a}[7] = 9$, etc. After executing `x = malloc(2);`, assigning `x->data = 12;`, the cell at index 7 is no longer free. It stores the data value 12. The next free cell becomes the first one available, i.e., we have $\mathtt{m} = 9$. After identifying the predecessor and successor

of x, i.e., inserting x into the sorted heap, we obtain the array show on the right in Fig. 7.

A call to `free(x)` pushes x onto the free list. That is, this call translates to a pair of statements `a[x] = m; m = x;`. The last freed cell will be the first free cell in the list of free cells, i.e., the subsequent call to `malloc` will return the last freed cell.

VII. Experimental results

In this section we discuss the results of our experiments with the proposed synthesis procedure on a number of real-world examples. The original input C programs and the resulting outputs are available at http://www.cs.cmu.edu/~jsimsa/c2vhdl.tar. Before discussing the outputs of our tool, we first describe the problems solved by the C-based software models.

Priority queue – This is our running example from Figure 1. The design has one input signal and one output signal. The implementation repeatedly inputs n elements, sorts them, and outputs them in a sorted order. For the sake of experimental evaluation we chose $n = 10$.

Merge sort – This example implements a merger of two sorted sequences. The design has two input signals and one output signal. The implementation repeatedly receives n_1 sorted elements through the first input signal and n_2 sorted elements through the second input signal. Using the merge sort it combines the two sequences into one sorted sequence, which is then output. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Packet sorting – This example implements a simple network element. The design has two input signals and one output signal. The implementation repeatedly inputs packet data through the first input signal and packet identifier through the second input signal. It inserts these packets into a buffer while ignoring duplicate identifiers, until it fills a buffer with n packets. It then sorts the received packets by their identifier and outputs them. For the sake of experimental evaluation we chose $n = 10$.

Binary search tree dictionary – This example implements a data structure for storing a set of elements with a test for membership. The design has two input signals and one output signal. The implementation repeatedly inputs n_1 elements through the first input signal and builds a binary search tree out of them. This is followed by receiving n_2 queries through the second input signal and producing the correct response through the output signal. For the sake of experimental evaluation we chose $n_1 = 10$ and $n_2 = 10$.

Each of these models was succesfully run through the sequence of procedures described in this paper: shape analysis, bounds analysis, and array transformation.

Program	Bound	C LOC	VHDL LOC	Bound synthesis time
merge	$8 * n_1 + 8 * n_2$	80	1927	600m
prio	$8 * n$	56	1475	4s
packet	$12 * n + 8$	95	2430	6s
bst_dict	$24 * n_1$	142	2703	80s

TABLE I
COMPUTED BOUNDS AND LINES OF CODE.

Program	ALUTs	Registers	Block Mem	Blocks	Speed
merge	5,157	4,694	8,192	2	90MHz
prio	5,859	4,598	4,096	1	83MHz
packet	9,413	9,158	8,192	2	76MHz
bst_dict	5,786	5,660	8,192	2	125MHz

TABLE II
SYNTHESIS AND IMPLEMENTATION RESULTS.

Table I lists the symbolic bounds for our examples in bytes.[2] These symbolic bounds were then concretized using the aforementioned values and run through our translation tool which inputs a C program and a concrete bound and generates a functionally equivalent VHDL program. Table I also lists lines of code (LOC) for both the hand-written C models and their automatically generated VHDL counterparts. The running time ranges from minutes to hours depending on the example.

Our VHDL generation step is carefully crafted to work well with FPGA synthesis tools. The generated VHDL files were synthesized using the Altera Quartus II 9.0 tools (build 184 04/29/2009 SP1 SJ Web Edition) targeting Stratix III FPGAs. The results are shown in Table II. The ALUT (Altera's adaptive look-up tables) column gives an indication of the size of the combinational elements in the generated design. The registers column indicates how many flip-flops in the logic fabric were used for registers. The block mem column indicates how many memory bits in the generated design were implemented using embedded memory blocks and the following column shows how many independent memories were synthesized. The last column shows the maximum speed. In all cases the tools automatically picked the smallest EP3SL50F484C2 FPGA and package and the timing results are given for this part.

Most of the synthesized circuits occupy only a small portion of the smallest Stratix-III FPGA. The largest design is packet which utilizes 25% of the combinational ALUTs but less than 1% of the available block memory and only 24% of the available logic registers. The smallest design is prio which occupies 15% of the available combinational ALUTs, 12% of the available logic registers and less than 1% of the available block memory. The operating frequency of these circuits is in a range which is typical for FPGA circuits used as co-processing circuits. We have tested several of our examples running on a Cyclone II FPGA on the Altera DE2 board. For example, the priority encoder circuit was synthesized, implemented and run on the Altera Cyclone II EP2C35F672C6 FPGA

[2]The size of data types and structure alignment of a 32-bit architecture (e.g. 4-byte pointers) is assumed.

(supporting 33,216 logic elements) and we have observed the correct behavior on actual hardware using the SignalTap logic analyzer. Our conclusion from these preliminary results is that we have identified a viable approach for translating heap-based C programs into VHDL designs which have acceptable area utilization and performance.

Our bounds computation algorithm was able to compute useful bounds. However, at the moment we do not have enough experimental data to provide an thorough estimate for the quality of bounds computation.

Examples of failure. Our approach for symbolic bounds synthesis can fail in various ways. For example, the input program might operate over DAGs (*e.g.* BDDs) or hash tables; in which case, we would currently fail to produce an arithmetic abstraction. Note that—even in the case of programs with simple linked data structures—improving the scalability and accuracy of shape analysis is an area of active research. When we successfully generate arithmetic abstractions, our constraint-based synthesis algorithm can also fail. The abstraction may be too coarse, or the problem may be too complex (*e.g.* highly non-linear). Consider the case of a "watcher list" for a literal ℓ in a SAT solver, which tracks the clauses in the clause database in which ℓ appears. A bound on the size of this list certainly exists, but our tool cannot work out what this bound is.

VIII. Conclusion

C-to-gates synthesis aims to bring together the agility of software development with the speed of raw gates. Until now, C-to-gates synthesis systems were lacking support for non-trivial dynamic allocation and deallocation on the heap, thus limiting the wider applicability of these tools. This paper has introduced a new method that synthesizes symbolic heap bounds expressed on generic parameters. The method uses computed shape invariants and abstractions together with a constraint solving based approach to find a symbolic expression representing the bound. Our system facilitates the use of common software abstractions and libraries (potentially with no memory bounds) within C-to-gates synthesis systems. Thus, designers can potentially use high-level abstractions (*e.g.* dynamically allocated trees and lists) when designing circuits.

References

[1] E. Albert, S. Genaim, and M. Gomez-Zamalloa. Heap space analysis for Java bytecode. In *ISMM*, 2007.

[2] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM Press, 2007.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ICFP*, 1998.

[4] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 2006.

[5] B. C. Brock, W. A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. *Technical Report 86,* `http://www.cli.com`, 1994.

[6] F. Bruschi and F. Ferrandi. Synthesis of complex control structures from behavioral SystemC models. *DATE*, 2003.

[7] B. A. Buyukkurt, Z. Guo, and W. Najjar. Impact of loop unrolling on throughput, area and clock frequency in ROCCC: C to VHDL compiler for FPGAs. *Int. Workshop On Applied Reconfigurable Computing*, 2006.

[8] B. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[9] M. A. Colon, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[10] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. *TACAS*, 2006.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[12] J. Farkas. Uber die theorie der einfachen ungleichungen. *Journal fur die Reine und Angewandte Mathematik*, 124:1–27, 1902.

[13] M. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. *FCCM*, 2000.

[14] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[15] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. *TACAS*, 2009.

[16] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *CAV*, 2009.

[17] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, 14, Apr. 1997.

[18] S. Gupta, N. D. Dutt, R. K. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. *VLSI Conference*, 2003.

[19] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, 2003.

[20] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In *ESOP*, 2006.

[21] IMEC. CleanC analysis tools. `http://www.imec.be/CleanC/`, 2008.

[22] R. Iosif, M. Bozga, A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.

[23] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. *SAS*, 2000.

[24] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. *SAS*, 2006.

[25] S. Magill, M. Tsai, P. Lee, and Y. Tsay. THOR: A tool for reasoning about shape and arithmetic. *CAV*, 2008.

[26] W. A. Najjar, A. P. W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 36(8), 2003.

[27] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

[28] A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, 2007.

[29] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. In *ESOP*, 2009.

[30] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard: a Scheme to hardware compiler. In *Workshop on Scheme and Functional Programming*, 2006.

[31] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear-relations analysis. *SAS*, 2004.

[32] L. Semeria and G. D. Micheli. SpC: synthesis of pointers in C. *ICCAD*, 1998.

[33] A. Takach, B. Bower, and T. Bollaert. C based hardware design for wireless applications. *DATE*, 2005.

[34] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, Y. Lu, and S. Vassiliadis. DWARV: Delftworkbench automated reconfigurable VHDL generator. *FPL*, 2007.

Author Index